型システム入門
プログラミング言語と型の理論

Benjamin C. Pierce 著

Types and Programming Languages

住井英二郎 監訳
遠藤侑介・酒井政裕・今井敬吾・黒木裕介・今井宜洋・才川隆文・今井健男 共訳

Ohmsha

日本語版に寄せて

“Types and Programming Languages”の日本語訳の出版を心よりお祝いいたします。1930年代以来、プログラミング言語コミュニティにおける私の同僚、指導教員、先人たちにより発展してきた数々の美しい考え方が、現代日本の研究者・技術者の皆様に幅広く読まれることをうれしく思うと共に、それを可能にした訳者の皆様（多くは現役のソフトウェア技術者）への感謝の意を、読者の皆様も私と共有していただければ幸いです。実際、彼らの努力により、日本語版では原書の誤植や技術的な誤りが数多く訂正されています。

“Types and Programming Languages”がはじめて出版されてからの12年間、本書が扱っている考え方はますます主流となり、今や多くの大学の学部のカリキュラムにおいて取り上げられていると共に、さまざまな学会においてプログラミング言語に関する着想を探求・伝達するための共通の概念・枠組みとして用いられています。同時に、HaskellやScalaのように、強力な型システムを備えた新しい言語が、産業界において大いに関心を集めています。「型」は今や黄金時代を迎えているのです。

2013年3月

Benjamin C. Pierce

監訳者序文

いきなりだが、出版までの経緯（私事を含む）や謝辞に興味のない方は、v ページの「実用的情報」だけでも参考にしていただければ幸いである。

さて、私が Benjamin C. Pierce 氏に出会ったのは、学部 4 年後期の研究室配属直後であった。Pierce 氏は TACS '97 という国際学会において "Comparing Object Encodings"（オブジェクト表現の比較）という論文発表 [62] を行うために日本を訪問中だったが、当時の私の所属研究室も訪問して、同じ発表を行った。それがちょうど本書の第 32 章（の発展）にあたる内容であった。氏の発表は私がはじめて聞く本格的研究発表でもあり、当然ながら内容を理解できるはずもなく、「ヨ型って何？」という状態だったが、研究室の先輩にそう聞いたところ、存在型（本書第 24 章参照）に関する論文 [325] を紹介された。それを端緒に Cardelli と Wegner のチュートリアル [88] をはじめ、単純型、多相型、再帰型、部分型、$F_{<:}$、F_ω そして $F^\omega_{<:}$ などの型システム、およびそれらの計算体系におけるオブジェクト表現に関するさまざまな論文を読んだ。その中でも Pierce 氏による一連の論文は、内容もさることながら、記号法などまで細心の注意を払った、定式化の美しさとプレゼンテーションのわかりやすさが強く印象に残った。

その数年後、私は Pierce 氏のご厚意により、延べ 3 年間にわたり米国で氏と研究を行う機会に恵まれた。そのちょうど 1 年目に Pierce 氏が自身の授業テキストを元に執筆し始め、2002 年に MIT Press から出版されたのが、本書 "Types and Programming Languages"（原題）、略して TAPL である。TAPL は、Pierce 氏の徹底した統一的定式化（序文にもある通り、本書の計算体系や型システムの定義は TinkerType というツールにより半自動的に合成されている）と、豊かな英語表現による丁寧な直観的説明（それが和訳においては逆に難題だったが）により、瞬く間に型システムの教科書（ひいてはプログラミング言語理論の入門書）の決定版となり、草稿段階から私の周囲だけでも何十回となく研究室輪講や大学院授業などに使われてきた。

そういった背景の中で、私の大学院研究室の同期であり、形式仕様記述言語 Alloy に関する本『抽象によるソフトウェア設計 — Alloy ではじめる形式手法』（オーム社、2011 年）も和訳した今井健男氏から、TAPL を和訳したいので Pierce 氏にコンタクトをとると共に、監訳者になってほしいとの依頼を受けたのは 2011 年 5 月のことだった。私自身は当初は「TAPL を英語ではなく日本語で読みたい」という層があまりイメージできず、実はやや懐疑的だったのだが、今井氏やオーム社の鹿野氏の熱意もあり引き受けることにした。結果的に、日本語版の発売が発表された際の Twitter での反応などを見ても、彼らの判断が正しかったことは明白である。

そのような熱狂的ともいえる反応の理由の一つは、この数年の（特に日本における）「関数型言語ブーム」や「形式手法ブーム」だろう。1990 年代前半頃までは「もはや C と C++ 以外は非現実的なのでプログラミング言語の研究は終わった」などとまことしやかに言う向きもあったようだが、Java やスクリプト言語の普及を経て、プログラミング言語の研究は「終わった」どころか、いまだかつてない盛況を呈している。POPL（ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages、プログラミング言語の原理に関する国際シンポジウム）や ICFP（ACM SIGPLAN International Conference on Functional Programming、関数型プログラミングに関する国際会議）といった国際学会、あるいは国内では PPL（日本ソフトウェア科学会プログラミング論研究会プログラミングおよびプログラミング言語ワークショップ）なども過去最高の参加者数を更新し続けている。しかも参加者は研究者や学生に限らず、企業技術者やフリープログラマも増えている。これはほんの 15 年前には想像できなかった状況で、隔世の感がある（この例からも、たかが数

年〜せいぜい数十年程度の短期的視野から、何らかの研究が「終わった」あるいは「非現実的」などと決めつけることがいかに有害かわかる）。

本書はプログラミング言語「理論」の基礎的教科書であり、特定のプログラミング言語や開発手法の解説書などに比べると「直ちに役立つ」ようには見えないかもしれない（もちろん、当該分野に興味のある学生や研究者、あるいは形式手法や関数型言語を用いる技術者であれば直ちに役立つ）。しかし、「操作的意味論」「型システム」あるいはそれらの基礎をなす「帰納的定義」「構造的帰納法」といった、プログラムすなわち「計算」の「理論的な見方」を学ぶことは、およそ現代人にとって論理的思考力が必須であるのと同様、システム設計・開発実務あるいは他の学習にも必ず役立つだろう。

本書は原文で600ページを超える大著であり、訳者ならびに編集者の方々の貢献は「多大」などという言葉ではとても言い表せない。どの訳者の方も、本業の傍ら、膨大な時間を本書の翻訳に費やしていただいた（プロジェクト管理システムTracのチケット数は1400件以上、それに対するコメントなどは約10000件、バージョン管理システムSubversionのコミット数は3000件を超え、いずれもオーム社開発部の記録更新とのことである）。

それに加えて「さらに」特筆すべき貢献（の一部）をここに列挙したい。遠藤氏は、本業に加えRuby言語のコミッタとしても活躍される中、演習の解答などに多くの非自明な誤りを発見し、世界中で最も（中国語版やロシア語版の翻訳者、あるいは原著者よりも？）注意深くTAPLを読んだのではないかと思われる（この点は日本語版の他の訳者も同様で、原著正誤表になかった150件以上の修正をフィードバックした）。酒井氏は翻訳の質が非常に高く、監訳者からはほとんど取るに足らない事項をコメントするばかりであった。今井敬吾氏と今井宜洋氏は、小規模ながら国内随一の関数型言語ヘビーユーザである企業から、コミュニティへの日頃の貢献に加え、非常に多忙な中を本書の翻訳に参加していただいた。黒木氏は、専門分野は異なるものの、その違いや豊富なTEXの知識を生かした多くの貢献をされた。才川氏は学生でありながら深い見識により、しばしば独自の視点から鋭い指摘をされた。今井健男氏は、前述の通りAlloyに関する本に続きTAPLの和訳プロジェクトそのものを立ち上げただけでなく、1年半の長きにわたり粘り強く丁寧なプロジェクト管理をしてくださった。鹿野氏、高尾氏をはじめとするオーム社の方々には、MIT Pressとの契約をはじめ、プロジェクト管理や編集・整形など、出版に関わるすべての面で大変なお世話になった。

最後になってしまったが、以下の方々からは和訳の草稿に対し数多くの貴重なコメントをいただいた。いずれも超一流の開発者・研究者などでいらっしゃるが、すべての方を詳しくご紹介するとそれだけで一冊の本ができてしまうので、単なる列挙（50音順）でどうかご容赦いただきたい。阿部秀彦氏、五十嵐淳氏、石井大海氏、稲葉一浩氏、小笠原啓氏、尾上能之氏、ジャック・ガリグ氏、川中真耶氏、田中英行氏、田渕直氏、対馬かなえ氏、古瀬淳氏、増子萌氏、三好博之氏、森田直幸氏。ここに深く感謝する。

実用的情報

本書の対象である型システムおよびプログラミング言語理論の分野は、用語の和訳が完全に定着するほど古くはなく、日本人研究者も特に口頭では単なる片仮名英語を使ってしまうことが多いが（例えば「多相性」ではなく「ポリモーフィズム」など）、今回の翻訳では不自然でない範囲で日本語化を心がけた。485ページから訳語集がまとめられているので参考にされたい。

本書の内容に特に関係が深い日本語書籍としては、五十嵐淳氏による『プログラミング言語の基礎概念』（サイエンス社、2011年、`http://www.fos.kuis.kyoto-u.ac.jp/~igarashi/CoPL/`）が挙げられるだろう。帰納的定義や操作的意味論、単純型、ML多相型、型推論およびそれらの実装についてコンパクトにまとまっており、本書のある種のダイジェスト版としても参考になると思われる。大堀淳氏による『プログラミング言語の基礎理論』（共立出版、1997年）も内容は近いが、

残念ながら手に入りにくいようだ。萩谷昌己氏と西崎真也氏による『論理と計算のしくみ』（岩波書店、2007 年）は（本書のようなプログラミング言語よりも）論理学を、横内寛文氏の『プログラム意味論』（共立出版、1994 年）は（本書のような操作的意味論よりも）表示的意味論を、それぞれ中心とする日本語書籍である。

本書で実装に用いられているプログラミング言語 OCaml については、五十嵐淳氏による『プログラミング in OCaml～関数型プログラミングの基礎から GUI 構築まで～』(技術評論社、2007 年、`http://www.fos.kuis.kyoto-u.ac.jp/~igarashi/OCaml/`）や、浅井健一氏による『プログラミングの基礎』（サイエンス社、2007 年、`http://pllab.is.ocha.ac.jp/~asai/book/Top.html`）などの和書がある。後者は特に説明の丁寧さで定評があり、関数型言語（ないしプログラミングそのもの）の初心者にも勧められる。

本書より発展的な内容を扱った続編として、各テーマの専門家による共著 "Advanced Topics in Types and Programming Languages"（洋書、Pierce 氏編、MIT Press、2004 年）がある。また、本書よりも基本的かつ幅広いプログラミング言語理論を定理証明環境 Coq 上で定式化して解説したテキストが Web にて公開されており（`http://www.cis.upenn.edu/~bcpierce/sf/`）、有志により和訳されている（`http://proofcafe.org/sf/`）。

2013 年 3 月

住井英二郎

序文

型システムの研究、そして型理論的観点からのプログラミング言語の研究は、ソフトウェア工学、言語設計、高性能計算向けコンパイラの実装、セキュリティを主要な応用分野に持つ、活発な研究分野となった。本書は、この分野の基本的な定義、結果、技法の包括的入門である。

想定読者

本書は主に二種類の読者層を想定している。一方は、プログラミング言語と型理論を専門とする大学院生および研究者である。もう一方は、プログラミング言語理論の鍵となる概念に触れたい、計算機科学の全分野の大学院生および習熟度の高い学部生である。前者の読者層は本書により、この分野についてくまなく、かつ、すぐに研究論文を読めるようになるほど詳しく学習できる。後者の読者層には、多数の入門的な題材と豊富な例題、演習、事例を提供する。プログラミング言語の初級の大学院課程でも上級のセミナーでも、教科書として使用できる。

指針

第一の指針は、核となるトピックを**網羅**することである。その中には、操作的意味論の基本と関連する証明技法、型無しラムダ計算、単純型システム、全称型や存在型による多相性、型再構築、部分型付け、有界量化、再帰型、型演算子が含まれる。また、他の数々のトピックについても簡単に議論する。

次の主要な指針は**実用主義**である。本書はプログラミング言語における型システムの利用に専念する。そのため、表示的意味論など、型付きラムダ計算に関するより数学的な教科書が扱うであろう一部のトピックを犠牲にした。本書では値呼びのラムダ計算を計算の基礎として用いる。これは今日のプログラミング言語のほとんどと合致し、参照や例外といった、命令的プログラミングの構成要素に容易に拡張できる。それぞれの言語機能に対する主な関心事は、その機能を考える実用上の**動機**、その機能を含む言語の**安全性**を証明する技法、そして、その機能がもたらす**実装上の問題**、特に、型検査アルゴリズムの設計と分析である。

さらなる指針は、この分野の**多様性**を尊重することである。本書は数々の個性的なトピックと、幾つかのよく理解されている組み合わせを網羅するが、すべてを一緒くたに、一つの統一的な体系にまとめようとしたりはしない。一部のトピックについては統一的な表現が与えられている。例えば、何種類かある「矢印型」については、**純粋型システム**の表記を用いて統一的に、すっきりとまとめて扱うことができる。しかし、この分野は全体として、今でも急速に成長中であり、完全に体系化することはできない。

本書は、講義においても独習においても**使いやすいこと**を意識して書かれている。ほとんどの演習に対して完全な解答が用意されている。核となる定義は参照しやすいよう、自己完結した図にまとめられている。概念と体系との間の依存関係は可能な限り明示している。さらに、豊富な参考文献と索引で補完している。

本書を構成する上での最後の指針は、**誠実さ**である。本書で議論されている体系は（何かのついでに触れただけの幾つかを除いて）すべて実装されている。各章には例題を機械的に検査するために使う型検査器とインタプリタが付属している。これらの実装は本書のウェブサイトから利用可能であり、プログラミングの演習、機能拡張の実験、大きめの授業課題に用いることができる。

以上の指針のために、犠牲になった望ましい性質もある。その中で最も重要なものは、網羅の**完全性**である。プログラミング言語と型システムの分野すべてをサーベイし尽くすことは、1冊の書籍の中ではおそらく（それが教科書なら確実に）無理だろう。本書の焦点は、核となる概念を注意

深く解説することである。そしてさらなる学習のために、その出発点となる研究論文への参照を多数掲載している。もう一つ、本書の指針にないものは、型検査アルゴリズムの実用上の**効率性**である。本書は業務用のコンパイラや型検査器の実装に関する本ではない。

構成

本書の第 I 部では型無しの計算体系を議論する。抽象構文や、帰納的定義と証明、推論規則、操作的意味論の基本的な概念を、数とブール値からなる非常に単純な言語を用いて最初に導入する。その後、型無しラムダ計算を同様に導入する。第 II 部では、単純型付きラムダ計算と、直積、和、レコード、バリアント、参照、例外といったさまざまな基本的言語機能を網羅する。型付き算術演算に関する予備的な章では、型安全性について鍵となるアイディアをやさしく紹介する。発展的内容として、単純型付きラムダ計算の正規化可能性に関する証明を、Tait の手法を用いて行う章もある。第 III 部では、部分型付けの基礎的な機構に取り組む。メタ理論[†1]と大きめの二つの事例が含まれる。第 IV 部は再帰型を、単純な**同型再帰**と少し難しい**同値再帰**の二つの定式化で取り扱う。その部の二つの章の後者では、同値再帰型と部分型付けのある体系のメタ理論を、余帰納法という数学的枠組みの中で解説する。第 V 部では多相性を取り上げる。各章では、ML 風の型再構築、System F の強力な非可述的多相、存在量化、および存在量化と抽象データ型との関係、そして、有界量化を伴う型システムにおける多相性と部分型付けの組み合わせを扱う。第 VI 部では型演算子を扱う。最初の章で基本的な概念を取り扱う。次の章では System F_ω とそのメタ理論を解説する。その次では型演算子と有界量化を組み合わせ、System $F^\omega_{<:}$ を構築する。最後は事例で終わる。

章の間の主要な依存関係を図 P-1 に示す。灰色の矢印は、後の章の一部だけが前の章に依存することを示す。

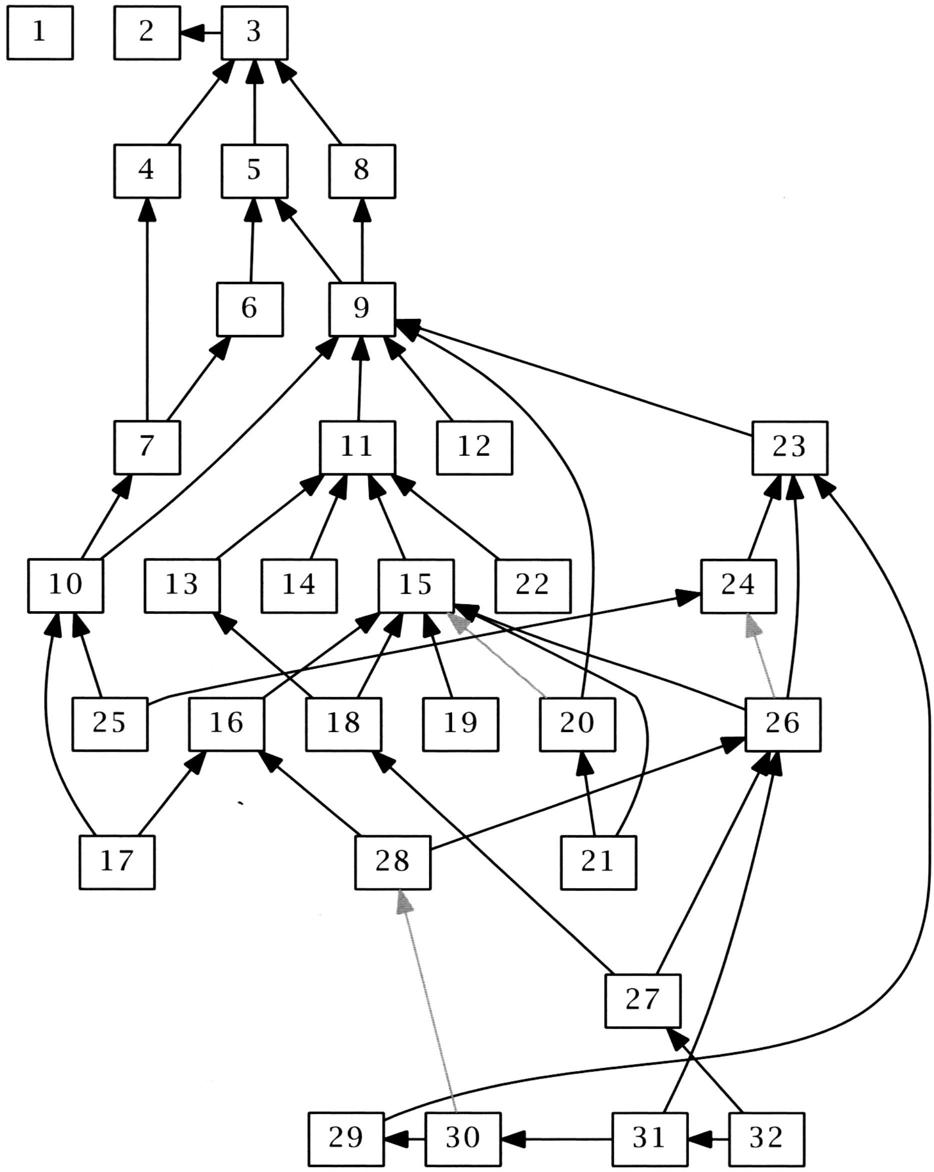
本書で議論する言語機能は、すべて共通のパターンに従って扱う。まず動機付けの例を示し、次に形式的な定義、それから型安全性などの基本的性質の証明を示す。そして（通常は別の章で）型検査アルゴリズムや、その健全性、完全性、停止性の証明につながるメタ理論をより深く分析し、最後に（これも別の章で）OCaml プログラムとしてこれらのアルゴリズムを具体的に実現する。

本書を通して、オブジェクト指向プログラミングの機能の分析と設計から多数の例を得ている。四つある事例の章では、異なるアプローチを詳しく説明する。それぞれ、従来の命令的オブジェクトとクラスの単純なモデル（第 18 章）、Java に基づく核となる計算（第 19 章）、有界量化を用いて命令的オブジェクトの説明を詳細化するアプローチ（第 27 章）、System $F^\omega_{<:}$ の純粋関数的な世界において、存在型を用いてオブジェクトとクラスを扱うアプローチ（第 32 章）である。

本書を上級課程の 1 学期分で扱える程度に小さくするため（そして平均的な大学院生でも扱える程度に軽くするため）、興味深く重要な話題を少なからず割愛せざるをえなかった。表示的・公理的意味論の扱いは完全に省略した。これらを扱う素晴らしい書物は既にあり、また、これらに取り組むと本書の強い実利的・実装指向の観点を損なってしまうだろう。型システムと論理学の間の密接な関係は数か所で触れてはいるが、詳しい追究はしていない。この話題は重要ではあるが、話が大きくそれてしまう。例えば、依存型、交差型、Curry-Howard 対応など、プログラミング言語と型システムの発展的な機能の多くは余談として言及しているだけである。これらの話題を扱う短い節に、参考文献の出発点を示してある。最後の点としては、Java 風の核言語の概覧（第 19 章）を除けば、本書は全面的にラムダ計算に基づいた計算体系に焦点を絞っている。しかし、この体系の下で説明した概念と仕組みは、関連する分野に直接応用できる。例えば、型付き並行言語、型付きアセンブリ言語、オブジェクトに特化した計算体系などである。

---

†1（訳注）何らかの理論的体系そのものの性質に関する理論のこと（18 ページ参照）。



▲ 図 P-1. 章の依存関係

必要な背景知識

本書はプログラミング言語理論の予備知識は前提としないが、読者は数学、特に離散数学、アルゴリズム、初等的な論理学に関する、学部レベルの論理的に正確な授業の内容に、ある程度習熟している必要がある。

読者は少なくとも一つの高階関数型プログラミング言語（Scheme、ML、Haskell など）と、プログラミング言語およびコンパイラの基礎的な概念（抽象構文、BNF 文法、評価、抽象機械など）に慣れている必要がある。この内容については、学部生向けの素晴らしい教科書が豊富にある。特に、Friedman、Wand、Haynes による “*Essentials of Programming Languages*” [163] や、Scott による “*Programming Language Pragmatics*” [430] がよいと思う。Java [23] などのオブジェクト指向

言語の経験があると幾つかの章で有用である。

型検査器の具体的な実装に関する章では、OCaml（ML のよく知られた方言）で書かれたコード断片が数多く掲載されている。これらの章は OCaml の事前知識があると読みやすいが、必ずしも必須ではない。この言語の一部しか使用しておらず、各機能についても初出の時点で説明している。これらの章は本書の中で、話の流れとして独立しているので、望むなら完全に飛ばして読み進むこともできる。

現時点で OCaml に関する最良の教科書は Cousineau と Mauny によるもの [120] である。OCaml の配布物（`http://caml.inria.fr` や `http://www.ocaml.org` で利用可能）に付属するチュートリアルも大変読みやすい。

ML のもう一つの著名な方言である Standard ML に慣れている読者は、OCaml のコード断片を読むのは難しくないだろう。Standard ML に関するよく知られた教科書は、Paulson [353] や Ullman [470] などがある。

授業の進め方

中級・上級の大学院課程なら、本書のほとんどを 1 学期で扱えるはずだ。図 P-2 は、Pennsylvania 大学の博士課程学生を対象とした上級講座のシラバスの例である。（週に 90 分の講義が二回。プログラミング言語理論についての最低限の事前知識しか仮定していない。ただし進行は速めである。）

学部課程や初級の大学院課程では、いろいろな進め方がある。「プログラミングにおける型システム」という授業なら、さまざまな型付け機能を紹介する章と、その使い方を示す章に集中的に取り組み、メタ理論や実装の章のほとんどは省略するとよいだろう。そうではなく、「型システムの基礎理論と実装」という授業なら、おそらく第 12 章（や、おそらく第 18 章や第 21 章）以外の前半の章をすべて読み進め、後半の発展的な内容を捨てるとよいだろう。より短い授業科目を構成する場合でも、図 P-1 に示す依存関係グラフを参考に、興味のある特定の章だけ選択すればよい。

本書は、プログラミング言語理論全般に関する大学院課程の教科書としても適切である。1 学期の半分から 3 分の 2 を費やして本書の推奨部分に取り組み、残りは、例えば Milner の π 計算の教科書（Milner [311]）に基づいて並行性の理論の単元にあてたり、Hoare 論理や公理的意味論（例えば Winskel [496]）の導入、継続やモジュールシステムなどの発展的な言語機能のサーベイについて取り組んだりするとよい。

演習プロジェクトが重要な役割を果たす授業ならば、理論的な内容（例えば正規化や、メタ理論の章の幾つか）は一部を後回しにして、学生がプロジェクトのテーマを選ぶ前にさまざまな例に触れておくことが望ましいかもしれない。

演習

ほとんどの章には多数の演習がある。紙と鉛筆でできるもの、議論している体系の**一部を使った**プログラミング例を扱うもの、その体系**全体の** ML 実装に拡張を施すものなどがある。各演習には以下の階級でおよその難易度を付記している。

★	確認テスト	30 秒から 5 分
★★	簡単	≤ 1 時間
★★★	中程度	≤ 3 時間
★★★★	難しい	> 3 時間

★の演習は、重要な概念についてその場で確認することを意図している。これらの演習を見かけたら読み進む前に少し立ち止まることを、読者に強く推奨する。各章では、だいたい宿題に出せる程

講義	内容	本書の該当箇所
第 1 回	課程の概要、歴史、事務連絡	第 1 章、(第 2 章)
第 2 回	事前準備：構文、操作的意味論	第 3 章、第 4 章
第 3 回	ラムダ計算の導入	5.1 節、5.2 節
第 4 回	ラムダ計算の形式化	5.3 節、第 6 章、第 7 章
第 5 回	型、単純型付きラムダ計算	第 8 章、第 9 章、第 10 章
第 6 回	単純な拡張、派生形式	第 11 章
第 7 回	さらなる拡張	第 11 章
第 8 回	正規化	第 12 章
第 9 回	参照、例外	第 13 章、第 14 章
第 10 回	部分型付け	第 15 章
第 11 回	部分型付けのメタ理論	第 16 章、第 17 章
第 12 回	命令的オブジェクト	第 18 章
第 13 回	Featherweight Java	第 19 章
第 14 回	再帰型	第 20 章
第 15 回	再帰型のメタ理論	第 21 章
第 16 回	再帰型のメタ理論	第 21 章
第 17 回	型再構築	第 22 章
第 18 回	全称多相	第 23 章
第 19 回	存在多相、ADT	第 24 章、(第 25 章)
第 20 回	有界量化	第 26 章、第 27 章
第 21 回	有界量化のメタ理論	第 28 章
第 22 回	型演算子	第 29 章
第 23 回	F_ω のメタ理論	第 30 章
第 24 回	高階部分型付け	第 31 章
第 25 回	純粋関数型オブジェクト	第 32 章
第 26 回	予備	

▲ 図 P-2.　上級の大学院課程でのシラバス例

度の量の演習が「推奨」とラベル付けされている。

ほとんどの演習は、完全な解答が付録 A に掲載されている。解答がない演習の解答を探して読者が落胆することがないよう、そのような演習には ↛ のマークを付けている。

印刷上の表記

ほとんどの章は、ある型システムの機能について緩く導入し、それからその型システムを、推論規則の集まりとして一つ以上の図に形式的に定義する。この定義は通常、参照しやすいように、そのとき議論している機能のための新しい規則だけでなく、計算体系を構成するのに必要なすべての規則を省略せずに示す。新しい部分は網掛けにして、以前の型システムとの差分を視覚的にわかりやすくしている。

本書製作において特徴的な点は、すべての例が組版の際に機械的に型検査されていることである。スクリプトで各章を走査して例を抽出し、議論している機能を含む専用の型検査器を生成・ビルドし、その検査器を抽出した例に適用した結果の出力を本文に挿入している。この処理の難しいところを担当するシステムは **TinkerType** といい、Michael Levin と私が開発した（Levin and

Pierce [286])。この研究は、全米科学財団による助成金 CCR-9701826 “*Principled Foundations for Programming with Objects*”、および CCR-9912352 “*Modular Type Systems*” の支援を受けた。

電子的な情報

本書のウェブサイトは以下の URL で参照できる。

`http://www.cis.upenn.edu/~bcpierce/tapl`

このサイトでは本文の正誤表、授業計画のための参考情報、読者が提供してくれた追加資料、本書の各章で扱った計算体系の実装（型検査器と簡単なインタプリタ）などの情報を提供している。

これらの実装は本書の例を試してみたり演習の解答をテストしたりするための環境として使える。読みやすく修正しやすいように洗練もされていて、簡単な実装演習から大規模な授業計画までの基礎として、私の授業で学生らに成功裏に用いられてきた。実装は OCaml で書かれている。OCaml コンパイラは無償で `http://caml.inria.fr` より入手でき、ほとんどの環境で非常に容易にインストールできる。

読者は Types Forum も知っておくべきだ。型システムやその応用についてのあらゆる話題を取り扱うメーリングリストである。このメーリングリストはあまり流量が多くならないよう、また、告知と議論の S/N 比が高く保たれるよう管理されている。アーカイブや加入の方法は `http://www.cis.upenn.edu/~bcpierce/types` にある[†2]。

謝辞

本書に価値を見出した読者は、Luca Cardelli、Bob Harper、Robin Milner、John Reynolds という 4 人の研究指導者に最大限の感謝をしてよいだろう。彼らは私に、プログラミング言語と型について私の知っていることのほとんどを教えてくれた。

他の知識のほとんども、共同研究を通して学んできた。Luca、Bob、Robin、John 以外の共同研究者としては、Martín Abadi、Gordon Plotkin、Randy Pollack、David N. Turner、Didier Rémy、Davide Sangiorgi、Adriana Compagnoni、Martin Hofmann、Giuseppe Castagna、Martin Steffen、Kim Bruce、小林直樹、細谷晴夫、五十嵐淳、Philip Wadler、Peter Buneman、Vladimir Gapeyev、Michael Levin、Peter Sewell、Jérôme Vouillon、住井英二郎がいる。これらの共同研究は、このテーマに関する理解の礎というだけでなく、楽しみの礎でもあった。

本書の構造と構成は、説明方法に関する Thorsten Altenkirch、Bob Harper、John Reynolds との議論により改良された。また、文章自体は Jim Alexander、Penny Anderson、Josh Berdine、Tony Bonner、John Tang Boyland、Dave Clarke、Diego Dainese、Olivier Danvy、Matthew Davis、Vladimir Gapeyev、Bob Harper、Erik Hilsdale、細谷晴夫、五十嵐淳、Robert Irwin、伊藤貴康、Assaf Kfoury、Michael Levin、Vassily Litvinov、Pablo López Olivas、Dave MacQueen、Narciso Martí-Oliet、Philippe Meunier、Robin Milner、Matti Nykänen、Gordon Plotkin、John Prevost、Fermín Reig、Didier Rémy、John Reynolds、James Riely、Ohad Rodeh、Jürgen Schlegelmilch、Alan Schmitt、Andrew Schoonmaker、Olin Shivers、Perdita Stevens、Chris Stone、住井英二郎、Val Tannen、Jérôme Vouillon、Philip Wadler の修正やコメントにより改良された（うっかりリストから誰かを漏らしていたら申し訳ない）。Luca Cardelli、Roger Hindley、Dave MacQueen、John Reynolds、Jonathan Seldin は、歴史に関する一部の錯綜した点について、当時のことを知る者としての視点を与えてくれた。

[†2] （訳注）現在は `http://lists.seas.upenn.edu/mailman/listinfo/types-list` に移動している。

1997 年、1998 年の Indiana 大学と、1999 年、2000 年の Pennsylvania 大学における大学院のセミナーの参加者は、本書の草稿の実験台となった。彼らの反応やコメントは、本書をこのように形作るにあたり、非常に参考になった。Bob Prior の率いる MIT Press のチームは、出版のプロセスの各段階でこの原稿をうまくとりまとめてくれた。原著のデザインは Christopher Manning が MIT Press のために開発した LaTeX のマクロに基づいている。

プログラムの証明は退屈すぎて、数学の社会的過程が機能しない。
—*Richard DeMillo, Richard Lipton, and Alan Perlis, 1979* [136]

……だから検証を社会的過程に頼ってはならない。
—*David Dill, 1999*

形式手法が大きな影響を与えることは決してない。それを理解しない人間でも使えるようになるまでは。
—*Tom Melham* の言とされる

目次

第 1 章

はじめに

1.1 計算機科学における型

現代のソフトウェア工学において、多種多様な**形式手法**は、システムが期待された動作の（明示的または暗黙の）仕様に照らして正しく振る舞うことを保証する手助けになるものと評価されている。広範に及ぶ形式手法の一端は、Hoare 論理、代数的仕様記述言語、様相論理、表示的意味論といった強力な枠組みである。これらは、ソフトウェアの多種多様な正しさを記述できるが、利用するのはしばしば煩雑で、プログラマ側にかなり高度な知識を要求する。一方、それよりずっと控えめであるが、控えめであるゆえに、自動検査器をコンパイラやリンカ、あるいはプログラム解析器に埋め込むことができ、背景となっている理論に馴染みのないプログラマでも使えるような技術がある。このような**軽量形式手法**のよく知られた例に、チップ設計や通信プロトコルといった有限状態システムのエラーを探索するツールである**モデル検査器**と呼ばれるツールがある。別の現在普及しつつある例としては**実行時モニタリング**と呼ばれる一連の技術があり、これらはシステムの一部が仕様通りに動いていないことを動的に検知する。しかし、今のところそれらより遥かに普及していて、最も確立された軽量形式手法は、本書で中心的に扱う**型システム**である。

「型システム」とは何か、プログラミング言語の設計者や開発者たちが日常的に口にする際の意味合いをすべて反映し、なおかつ十分に意味のある具体的な定義を与えるのが難しいのは、複数の大きなコミュニティが共有する多くの用語と同様である。しかし、次のような定義はできよう。

> **型システム**とは、プログラムの各部分を、それが計算する値の種類に沿って分類することにより、プログラムがある種の振る舞いを起こさないことを保証する、計算量的に扱いやすい構文的手法である。

幾つか補足すべき点がある。まず、この定義は型システムを、**プログラム**に関する推論のためのツールだとみなしている。このような用語の使い方は、本書が、プログラミング言語で用いられる型システムに関するものであるという方向性を反映している。より一般的には、**型システム**（あるいは**型理論**）という用語は、論理学や数学、哲学の、もっと広範な研究分野に言及するものである。この意味での型システムは、1900 年代初頭、Russell のパラドックス [424] など、数学の基盤を揺るがした論理的パラドックスを回避する手段として、最初に形式化された。20 世紀を通じて、型は論理学、特に証明論（Gandy [167] や Hindley [209] を見よ）において標準的に用いられる道具となり、また哲学や科学の言語として浸透していった。この分野における主要な成果として、原点である Russell の**分岐階型理論** [493]、Ramsey による**単純階型理論** [394] （これは Church の単純型付きラムダ計算 [100] の基礎となった）、Martin-Löf の**構成的型理論** [299; 301]、Berardi、Terlouw、Barendregt による**純粋型システム** [43; 458; 40] などがある。

計算機科学に限っても型システムの研究には二つの大きな分野がある。より実践的なほうはプログラミング言語への応用を扱う分野で、本書で中心的に扱うのもこちらの分野である。もう一方のより抽象的な分野では、**Curry-Howard 対応**（9.4 節）を通じて、さまざまな「純粋型付きラムダ計算」とさまざまな論理との間のつながりに着目する。どちらの分野においても同様の概念や表記法、技法が用いられるが、方向性で大きく異なる点が幾つかある。例えば、型付きラムダ計算の研

究では通常、正しく型付けされた計算はすべて停止することが保証されるような体系を考える。一方、ほとんどのプログラミング言語では、再帰的関数定義のような機能のため、停止性を犠牲にしている。

もう一つ上述の型システムの定義で重要なのは、実行時に計算される値の性質によって項（構文的表現）を**分類**することに重きが置かれている点である。型システムは、プログラム中の項が実行時にどう振る舞うかの**静的な**近似を求めるものとみなせる。（さらに言えば、項に関連付けられた型は、一般に式の型がその部分式の型にだけ依存するよう**合成的に**求められる。）

「静的」という語を明示的に付加することがある。例えば、「静的型付きプログラミング言語」という言い方をする。これは、本書で考察しているような種類のコンパイル時解析と、Scheme（Sussman and Steele [450]; Kelsey, Clinger, and Rees [252]; Dybvig [142]）のような言語で使われる**動的型付け**（あるいは**潜在的型付け**）とを区別するためである。後者ではヒープ中の異なる種類の構造を区別するのに実行時の**型タグ**が利用される。「動的型付けされる」といった言い回しは誤っているといって差し支えなく、おそらく「動的検査される」と言い換えるべきであるが、標準的に使われる用語法である。

静的であるがゆえに、型システムは必然的に**保守的**でもある。つまり、プログラムがある種の振る舞いを**起こさない**ことは断定的に証明できるが、起こすことは証明できない。それゆえ、時として実際には実行時に正しく振る舞うプログラムを拒絶してしまう。例えば、

```
if <複雑な条件> then 5 else <型エラー>
```

のようなプログラムは、正しく型付けできないとして拒絶されるだろう。たとえ `<複雑な条件>` が常に `true` に評価されるものであっても、静的解析ではそう判断できないからである。型システムの設計では、根本的に保守的性質と表現力との間の綱引きが避けられない。より多くのプログラムを（各部分により正確な型を与えて）型付けしたいという欲求が、この分野における研究の主要な牽引力である。

関連して、ほとんどの型システムで実現されている比較的単純な解析では、任意の望ましくないプログラムの挙動を排斥することはできない。正しく型付けされたプログラムには**ある**種の違反がない、ということしか保証できない。例えば、ほとんどの型システムでは、プリミティブな算術演算の引数が常に数であることや、メソッド呼び出し時にレシーバオブジェクトが要求されたメソッドを常に提供していることなどは静的に検査できるが、除算の二つ目の引数がゼロでない、あるいは配列へのアクセスが常に境界内に収まっている、などの検査はできない。

与えられた言語において型システムによって除去できる不正な振る舞いのことをしばしば**実行時型エラー**と呼ぶ。ここで留意すべきことは、何を不正な振る舞いとするかは、言語によって変わるという点である。実行時型エラーとされる振る舞いには、異なる言語同士で重なる部分も大きいが、どの型システムにも防ぎたい振る舞いの定義が付随する。各型システムの**安全性**（あるいは**健全性**）は、それぞれの実行時エラーの定義に照らし合わせて判断されねばならない。

型の解析で見つかる不正な振る舞いは、存在しないメソッドの呼び出しといった低水準な不具合に限らない。型システムはより高水準な**モジュール性**を強制することや、ユーザの定義した**抽象**の整合性を保護するのにも使われる。情報が正しく隠蔽されないこと、例えば内部表現が抽象的に扱われているデータのフィールドへの直接アクセスは実行時型エラーである。これは整数をポインタとして扱った結果、計算機をクラッシュさせるのと、全く同様である。

型検査器はコンパイラやリンカの中に構築されるのが一般的である。これはつまり、型検査器は**自動的に**、プログラマの介入や対話なしに動作しなければならないということである。すなわち、**計算量的に扱いやすい**解析でなければならない。しかしそれでも、プログラムに**型注釈**を明示することで、プログラマが手引きする余地もかなり残っている。こうした型注釈はプログラムの書きや

すさと読みやすさを考慮して比較的簡潔に保たれることが多い。しかし原理的には、プログラムが何らかの仕様を満たすことの完全な証明を型注釈で書き下すことも可能であろう。この場合、型検査器は事実上の**証明**検査器になる。拡張静的検査（Detlefs, Leino, Nelson, and Saxe [137]）といった技術は、型システムと、本格的なプログラム検証手法との間にそのような領域を設けようとしており、「適度に軽量な」プログラム注釈にのみ依存する、広い範囲の正当性を完全自動で検査する環境を実現する。

同じ理由から、原理上自動化可能であるだけでなく、型を検査する**効率的な**アルゴリズムを実際に備えている手法に最も興味がある。しかし、何をもって効率的とするかは議論の余地がある。MLの型システム（Damas and Milner [129]）のような広く使われているものですら、病的なケース（Henglein and Mairson [204]）では非常に長い時間を型検査に要する。型検査や型再構築が、決定不能であるが「実用上関心のあるほとんどの場合においては」素早く停止するアルゴリズムを備えた言語も存在する（Pierce and Turner [376]; Nadathur and Miller [333]; Pfenning [359]）。

1.2 型システムで何ができるか

エラーの検出

静的型検査の最もわかりやすい恩恵は、ある種のプログラミングのエラーを早期に検出できることである。早期に検出されたエラーはすぐに修正できるが、コード内に潜んでいて相当後に発見されるエラーはそうはいかない。その頃にはプログラマは別のことをしているし、プログラムの運用が開始された後の場合さえある。さらに、実行時よりも型検査時のほうがエラーの箇所を正確に特定しやすいことが多い。実行時では、何かおかしくなってからしばらく経った後でないと影響が見えてこないことがあるからだ。

実際、静的型検査は驚くほど広範囲のエラーを明らかにしてくれる。強力な型のある言語で開発を行うプログラマは、自分たちのプログラムがいったん型検査器を通れば「ちゃんと動く」傾向にあり、それが期待している以上によくあるとしばしば言う。考えられる説明としては、ただの不注意（平方根を取る前に文字列を数に変換し忘れる、など）だけでなく、もっと深い概念上の間違い（複雑な場合分けで境界条件を見落とす、あるいは科学計算で単位を勘違いする、など）が、しばしば型レベルの不整合となって現れる、ということが挙げられる。この効果の大きさは、型システムの表現力と取り組んでいるプログラミングの内容とに依存する。多様なデータ構造を扱うプログラム（コンパイラのような記号処理アプリケーションなど）は、科学系アプリケーションにおける数値計算のような数種類の単純な型しか使わないプログラムよりも型検査器から得られる効能が大きい（ただし、**次元解析**（Kennedy [253]）をサポートする詳細型システムは数値計算のようなプログラムにおいても極めて有用である）。

一般に、型システムの恩恵を最大限に享受するには、言語に用意された機構を活用する意思と併せてプログラマ側で意識すべきことがある。例えば、複雑なプログラムに現れるデータ構造をすべてリストで表現してしまうと、それらをそれぞれ異なるデータ型や抽象型で定義するのに比べて、コンパイラの援助は得られにくくなる。表現力の高い型システムは、データ構造に関する情報を型を使って表現するための数々の「仕掛け」を提供する。

ある種のプログラムにとって、型検査器はかけがえのない**保守**のためのツールともなる。例えば、複雑なデータ構造の定義を変更する必要があるとき、その構造に関連するソースコードで修正を要する箇所を探すのに大きなプログラム全体を手作業で調べる必要はない。データ型の定義を変更すれば、修正を要する箇所では必ず型の不整合が起こるので、コンパイラにかけて型検査が失敗する

箇所を調べるだけで列挙できる。

抽象化

型システムがプログラミングの工程に寄与する重要な点として、プログラミングの規律が強化されるということも挙げられる。特に大規模ソフトウェアを組み立てる際には、型システムは大規模システムのコンポーネントをパッケージ化し、つなぎ合わせる**モジュール言語**の基幹をなす。型はモジュール（あるいはクラスのような関連する構造）のインターフェイスに登場する。実際のところ、インターフェイス自身「モジュールの型」だとみなすことができ、モジュールが提供する機構の要約を提供する。これはモジュールの実装者と利用者との間の一種の契約ともいえる。

曖昧さのないインターフェイスを持つモジュールを用いて大規模システムを構成すれば、設計のスタイルがより抽象的になる。そのようなスタイルでは、インターフェイスを最終的な実装と切り離して設計、議論できる。インターフェイスをより抽象的に考えることは、一般に、より良い設計につながるものである。

ドキュメント化

型はプログラムを**読む**際にも有用である。手続きの先頭やモジュールインターフェイスの型宣言はある種のドキュメントを構成し、振る舞いを理解する有用なヒントになる。さらに、コンパイラが実行されるたびに検査されるので、コメント中に埋め込まれた記述とは違ってこのドキュメントは古くなることがない。型の役割は、モジュールのシグネチャにおいて特に重要である。

言語の安全性

「安全な言語」という言葉は、残念ながら「型システム」よりも議論を呼ぶ言葉である。字面でわかった気になる人も多いが、何をもって言語の安全性と考えるかは各人が属する言語のコミュニティから強く影響を受ける。とはいえ、平たく言えば、安全な言語とはプログラミングの最中にうっかり自分の足を撃つことが不可能な言語であると定義できる。

この直観的な定義を少し洗練すれば、**安全な言語とは自らがもたらす抽象を保護するものである**といえるだろう。いかなる高水準言語も、計算機が提供するサービスを抽象化する。安全性と言ったときに指し示しているのは、そのような抽象や、それより高位の抽象（プログラマが言語の機能を用いて定義するもの）の整合性を保証する、言語の能力である。例えばある言語で、計算機が備えるメモリの抽象として、配列の機能が参照と更新の操作付きで提供されているとしよう。この言語を使うプログラマは、配列が変更されうるのは明示的に更新の操作を行った場合のみであり、他のデータ構造の境界を越えてきたデータが書き込まれるようなことはないと期待するだろう。同様に、レキシカルスコープの変数はそのスコープ内でのみアクセス可能であると期待するし、呼び出しスタックは実際のスタックのような振る舞いをすると期待するだろう。計算機が提供するこうしたサービスの抽象を、安全な言語では**抽象的に**用いることができる。安全でない言語ではそれができず、プログラムがどう（誤）動作するかを完全に理解するのに、メモリ内のデータ構造のレイアウトやコンパイラが配置する順序といった低水準な詳細をプログラマがすべて意識していなければならない。極端な話、安全でない言語で書かれたプログラムは、そのプログラムのデータ構造を破壊しうるだけでなく、実行時システムのデータ構造も破壊しうる。その場合、何が起きても不思議ではない。

言語の安全性は静的型安全性と同じものではない。言語の安全性は静的型検査を使って**達成**しうるが、実行時検査により、おかしな操作を実行の瞬間に捕捉して、プログラムを止めるか例外を発

生させることにより達成することもできる。例えば、Scheme は静的型システムを持たないが安全な言語である。

一方、安全でない言語にも明らかな不注意の類いを除去するのに役立つ程度の「ベストエフォート」な静的型検査器が提供されることはよくあるが、我々の定義に従えばそのような言語は型安全であるとはいえない。正しく型付けされたプログラムが正しく振る舞うという**保証**を与えることが、一般的にはできないからである。それらの言語用の型検査器は、実行時型エラーが存在することを示唆してはくれるが（もちろん、何もしないよりはましである）、存在しないことを証明してはくれない。

	静的検査	動的検査
安全	ML、Haskell、Java など	Lisp、Scheme、Perl、PostScript など
安全でない	C、C++ など	

上の表で右下の欄が空白なのは、**ほとんどの**操作の安全性を実行時に強制する機構が既にあるなら、**すべての**操作を検査するために追加で必要なコストはほとんどないからである。（実際には、誤って用いれば実行時システムの整合性を破壊できるような、任意のメモリロケーションを読み書きできる低水準プリミティブを備えた動的検査言語も稀に存在する。例えば、最低限のオペレーティングシステムの上で動くマイクロコンピュータ向けの Basic の方言などである。）

実行時の安全性は通常、静的型付けだけでは達成できない。例えば、上記の表で安全だとした**すべての**言語は、**配列の境界検査**を動的に行う[†1]。同様に、静的検査言語であっても、あえて実際には健全ではない型検査規則を持つ操作（例えば Java のダウンキャスト操作。151 ページを見よ）を提供し、そうした構文が利用されている箇所を動的に検査することで言語の安全性を確保していることがある。

言語の安全性が絶対的であることは滅多にない。安全な言語がプログラマに「脱出用ハッチ」を準備していることは多い。安全でない可能性がある他の言語で書かれたコードを呼び出す外部関数呼び出しなどがそれにあたる。実際のところ、こうした脱出用ハッチが言語そのものの内部で制御された形で提供されることもある。OCaml（Leroy [280]）の `Obj.magic` や SML/NJ の `Unsafe.cast` などである。Modula-3（Cardelli et al. [83]; Nelson [338]）や C#（Wille [495]）はさらに進んで、ガベージコレクタといった低水準の実行時機構を実装することを意図した「安全でない部分言語」を提供している。この部分言語の特別な機能は、`unsafe` と明示的に印が付いたモジュールでしか使えないようになっている。

Cardelli [82] は少し違った観点から言語安全性についてまとめており、Cardelli が言うところの**捕捉される**実行時エラーと**捕捉されない**実行時エラーを区別している。捕捉されるエラーは計算を即座に止める（あるいは、そのプログラム内で楽に処理可能な例外を発生させる）が、捕捉されないエラーは計算を（少なくともしばらくの間）継続させる可能性がある。捕捉されないエラーの例としては、C のような言語で配列の境界を越えたデータアクセスがある。この観点においては、安全な言語とは、実行時に捕捉されないエラーを防止する言語のことである。

さらに別の観点では、可搬性に着目する。これは「安全な言語はプログラマ用マニュアルによって完全に定義される」というスローガンとして表現できる。ここで言語の**定義**とは、その言語で記述できるすべてのプログラムの振る舞いを予測するためにプログラマが理解しなければならない物事の集合だとしよう。すると、C のような言語のマニュアルは定義にならない。なぜなら、ある種

[†1] 配列の境界検査を静的になくしてしまうことは、型システム設計者の長年の目標である。原理的に必要な仕組み（**依存型**（30.5 節を見よ）に基づくもの）はよく理解されているが、表現力、型検査の予測可能性と計算量的な扱いやすさ、およびプログラム注釈の複雑さをうまく均衡させた形でその機能を言語に組み込むことは重要な課題として残されている。この分野に関する最近の進んだ研究成果は Xi and Pfenning [502; 503] に記載されている。

のプログラム（境界検査を行わない配列アクセスやポインタ算術が関与するもの）は、特定のCコンパイラによるデータ構造のメモリ配置といった詳細を知らなければ振る舞いを予測できず、別のコンパイラで実行すると、同じプログラムで振る舞いが全く異なる可能性があるからである。対照的に、Java、Scheme、MLのマニュアルは、その言語のすべてのプログラムの振る舞いの仕様を（厳密さの度合いは異なるが）正確に定める。正しく型付けされたプログラムは、これらの言語が正しく実装されている限り、常に同じ結果を返す。

効率性

計算機科学における最初の型システムは、1950年代初頭、Fortran（Backus [34]）などの言語で整数の算術式と実数の式とを区別し、数値演算の効率化を図るために導入された。型の導入により、コンパイラはプリミティブな演算に異なる表現を用い、適切な機械語命令を生成できるようになった。安全な言語では、安全性を保証するために必要な動的検査の多くを（常に成功することを静的に証明することで）除去し、さらなる効率改善を図ることができる。今日、ほとんどの高性能コンパイラは、最適化とコード生成のフェーズにおいて型検査器が収集する情報に大きく依存している。型システム**そのもの**は持たない言語のコンパイラにしても、多大な苦労をしてこの種の型情報を近似的に復元している。

型情報による効率改善は意外な部分からも得られることがある。例えば近年、型解析から生成される情報を使うことで、科学計算の並列プログラムではコード生成時の判断だけでなくポインタ表現も改善できることが示された。Titanium言語（Yelick et al. [506]）は、型推論の技術を用いてポインタのスコープを解析し、それをもとに、プログラマが明示的にプログラムを手動でチューニングするより有意に良い判断を可能としている。ML Kitコンパイラは強力な**リージョン推論**アルゴリズム（Gifford, Jouvelot, Lucassen, and Sheldon [176]; Jouvelot and Gifford [244]; Talpin and Jouvelot [454]; Tofte and Talpin [465; 466]; Tofte and Birkedal [463]）を使い、スタックベースのメモリ管理をすることで、ガベージコレクションをほとんど（プログラムによっては完全に）不要にしている。

その他の応用例

型システムは、プログラミングや言語設計における従来の利用方法の枠を越え、計算機科学やその関連領域においてさまざまな具体的方法で応用され始めている。ここでその一部を概説する。

型システムの応用分野として重要さが増しているのが、計算機やネットワークのセキュリティである。例えば、静的型付けはJavaとJINIのネットワーク機器向け「プラグアンドプレイ」アーキテクチャ（Arnold et al. [24]）のセキュリティモデルの核になっており、また証明付きコード（Proof-Carrying Code）（Necula and Lee [336; 337]; Necula [335]）を可能にする重要な技術である[†2]。同時に、セキュリティの分野で開拓された多くの基盤的なアイディアが、プログラミング言語の文脈で、しばしば型解析の形で再び研究の対象となろうとしている（例えばAbadi, Banerjee, Heintze, and Riecke [2]; Abadi [1]; Leroy and Rouaix [283]など）。逆に、プログラミング言語理論をセキュリティの領域の問題（例えばAbadi [1]やSumii and Pierce [449]）に直接応用することが注目を浴びつつある。

型検査・推論のアルゴリズムはコンパイラ以外のプログラム解析ツールでも多用されている。例えばAnnoDominiはCOBOLプログラムの2000年問題対応変換ユーティリティだが、ML風の型推

[†2] 著者からの補足：より最近の例としては、C#およびCLR、Andrew Myersによるセキュリティ型付きJava（JFlow、JIF、SIF、Fabric）、およびMicrosoft社のF*などがある。

論エンジン（Eidorff et al. [143]）がもとになっている。型推論の技術は別名解析（O'Callahan and Jackson [340]） や例外解析（Leroy and Pessaux [282]）のためのツールにも使われている。

自動定理証明においては、論理的性質や証明を表現するために型システム（通常は依存型に基づく非常に強力なもの）が使われる。NuPRL（Constable et al. [110]）、Lego（Luo and Pollack [291]; Pollack [390]）、Coq（Barras et al. [41]）、Alf（Magnusson and Nordström [297]）といった、ポピュラーな対話的証明系は型理論を直接用いている。Constable [109] と Pfenning [361] で、これらの体系の歴史を議論している。

データベースの分野においても、Document Type Definition（文書型定義、DTD）（XML 1998 [504]）や他のスキーマ（XML Schema 標準（XS 2000 [505]）など）といった、構造化されたデータを XML で記述する「ウェブ・メタデータ」の爆発的な広まりと共に、型システムへの興味が増加傾向にある。XML へのクエリ・操作を行う新しい言語は、これらのスキーマ言語に直接基づいた、強力な静的型システムを提供している（Hosoya and Pierce [221]; Hosoya, Vouillon, and Pierce [222]; Hosoya and Pierce [219]; Relax [398]; Shields [433]）。

型システムの、かなり毛色の異なる応用例として、計算言語学の分野に登場するものがある。型付きラムダ計算は**範疇文法**（van Benthem [473]; van Benthem and Meulen [474]; Ranta [395] など）の形式化の基礎をなす。

1.3 型システムと言語設計

型検査を考慮して設計されていない言語に型システムを組み込むのは困難な問題である。理想的には、言語設計は型システムの設計と手を取り合って進めるべきである。

その理由の一つとしては、型システムのない言語では、たとえ安全な動的検査言語であっても、型検査を困難あるいは不可能にする機能が提供されていたり、そうしたプログラミングイディオムが推奨されたりする傾向にある点が挙げられる。実際、型付き言語では型システムそのものが設計の基盤であり、他のすべての側面がそれを踏まえて考慮される、中心的原理であるとされることもしばしばある。

もう一つの理由として、型付き言語の具象構文は型注釈の考慮が必要なため、型無し言語より複雑になる傾向がある、という点がある。きれいで包括的な構文設計は、すべての課題に同時に対処するほうがうまくいきやすい。

型がプログラミング言語にとって不可欠であるという主張は、プログラマが型注釈を物理的に書き下すべき場所はどこか、代わりにコンパイラによる推論が可能な場所はどこか、という議論とは別である。きちんと設計された静的型付き言語であれば、膨大な型情報の明示的で退屈な維持管理をプログラマに強いることは決してない。とはいえ、どの程度の量の型情報を過剰と考えるかは議論の余地がある。ML 系言語の設計者は、必要な情報の復元に型推論を使うことで、注釈が最小限で済むように大変な労力を費やしている。Java を含む C 系の言語は、もう少し冗長な流儀を選択している。

1.4 歴史の概要

計算機科学において、最初期の型システムは、(例えばFortranにおける)整数と浮動小数点数の間の、非常に単純な区別をするために使われた。1950年代終わりから1960年代初期にかけて、この分類はデータ構造(レコードの配列など)と高階関数にまで拡張された。1970年代においては、幾つかのより高度な概念(パラメータ多相、抽象データ型、モジュールシステム、部分型付け)が導入され、型システムはそれ自身が一つの研究分野となった。同時に計算機科学者は、プログラミング言語に見られる型システムと数理論理学で研究されてきたそれとの間の関係に気づき始め、現在へと至る豊富な相互作用につながっている。

図1-1は、計算機科学における型システムの歴史で重要な点を短く(そして恥ずかしいほど不完全に)まとめた年表である。関連する論理学の発展についてはアスタリスク(*)付きで記述しており、この分野での貢献がいかに重要かを示すようにしている。右側にある参考文献は巻末の一覧で確認できる。

1.5 関連する文献

本書は自己完結することを目指してはいるが、すべてを包括しているとはとてもいえない。この分野は大変広く、多くの観点からのアプローチが可能なため、一冊の本だけで十分には取り扱えない。本節では良い入門となる資料を列挙する。

Cardelliによるハンドブックの記事[82]やMitchellによるもの[321]は、この分野への手軽な入門である。Barendregtの記事[40]は数学的に少し高度である。Mitchellの分厚い教科書"*Foundations for Programming Languages*"[322]はラムダ計算の基本と、さまざまな型システム、意味論の多くの観点を押さえている。実装上の問題よりは意味論に重点が置かれている。Reynoldsの"*Theories of Programming Languages*"[420]は大学院レベルのプログラミング言語理論のサーベイで、多相性、部分型付け、交差型を見事に解説している。Schmidtによる"*The Structure of Typed Programming Languages*"[428]は言語設計の文脈で型システムの核となる概念を解説している。幾つかの章では従来的な手続き型言語も対象にしている。Hindleyの"*Basic Simple Type Theory*"[209]は型付きラムダ計算と、それに密接にかかわる体系に関する結果をうまくまとめている。内容は広範というより深く掘り下げられている。

AbadiとCardelliの"*A Theory of Objects*"[4]は、本書と同じ題材を多く扱っているが、実装の側面にはあまり重きをおかず、代わりにオブジェクト指向プログラミングの基礎的な取り扱いにおけるこれらのアイディアの応用に集中している。Kim Bruceの"*Foundations of Object-Oriented Languages: Types and Semantics*"[60]は同様の問題を扱う。オブジェクト指向型システムの入門的な資料はPalsberg and Schwartzbach[350]とCastagna[92]にも含まれている。

型無し言語、型付き言語の双方に通じる意味論的な基礎付けは、Gunter[188]やWinskel[496]、Mitchell[322]といった教科書で詳細に扱っている。操作的意味論はHennessy[205]でも詳細に扱っている。**圏論**という数学的枠組みの中で型の意味論の基礎を扱う文献は多数あり、Jacobs[235]や、Asperti and Longo[25]、Crole[125]などがある。その簡単な入門として、"*Basic Category Theory for Computer Scientists*"[364]がある。

Girard、Lafont、Taylorによる"*Proofs and Types*"[179]は型システムの論理学的な側面(Curry-Howard対応など)を扱っている。その中ではSystem Fを創造者自身が説明している。線形論理を紹介する付録もある。型と論理のつながりはPfenningの"*Computation and Deduction*"でさらに読み進むことができる。Thompsonの"*Type Theory and Functional Programming*"[460]とTurnerの"*Constructive Foundations for Functional Languages*"[469]は関数的プログラミング

1870 年代	*形式論理学の起こり	Frege, 1879 [161]
1900 年代	*数学の形式化	Whitehead and Russell, 1910 [493]
1930 年代	*型無しラムダ計算	Church, 1941 [101]
1940 年代	*単純型付きラムダ計算	Church, 1940 [100]; Curry and Feys, 1958 [128]
1950 年代	Fortran	Backus, 1981 [34]
	Algol-60	Naur et al., 1963 [334]
1960 年代	*Automath プロジェクト	de Bruijn, 1980 [135]
	Simula	Birtwistle et al., 1979 [46]
	*Curry-Howard 対応	Howard, 1980 [224]
	Algol-68	van Wijngaarden et al., 1975 [476]
1970 年代	Pascal	Wirth, 1971 [497]
	*Martin-Löf 型理論	Martin-Löf, 1973 [299], 1982 [300]
	*System F、F^{ω}	Girard, 1972 [177]
	多相ラムダ計算	Reynolds, 1974 [408]
	CLU	Liskov et al., 1981 [288]
	多相型推論	Milner, 1978 [307]; Damas and Milner, 1982 [129]
	ML	Gordon, Milner, and Wadsworth, 1979 [183]
	*交差型	Coppo and Dezani-Ciancaglini, 1978 [115]; Coppo, Dezani-Ciancaglini, and Sallé, 1979 [116]; Pottinger, 1980 [392]
1980 年代	NuPRL プロジェクト	Constable et al., 1986 [110]
	部分型付け	Reynolds, 1980 [411]; Cardelli, 1984 [73]; Mitchell, 1984 [317]
	存在型としての ADT	Mitchell and Plotkin, 1988 [325]
	*Calculus of Constructions	Coquand, 1985 [117]; Coquand and Huet, 1988 [118]
	*線形論理	Girard, 1987 [178]; Girard et al., 1989 [179]
	有界量化	Cardelli and Wegner, 1985 [88]; Curien and Ghelli, 1992 [127]; Cardelli et al., 1994 [86]
	*Edinburgh Logical Framework	Harper, Honsell, and Plotkin, 1992 [195]
	Forsythe	Reynolds, 1988 [415]
	*純粋型システム	Terlouw, 1989 [458]; Berardi, 1988 [43]; Barendregt, 1991 [39]
	依存型とモジュール性	Burstall and Lampson, 1984 [68]; MacQueen, 1986 [294]
	Quest	Cardelli, 1991 [79]
	作用システム	Gifford et al., 1987 [176]; Talpin and Jouvelot, 1992 [454]
	列変数、拡張可能レコード	Wand, 1987 [487]; Rémy, 1989 [399]; Cardelli and Mitchell, 1991 [87]
1990 年代	高階部分型付け	Cardelli, 1990 [78]; Cardelli and Longo, 1991 [85]
	型付き中間言語	Tarditi, Morrisett, et al. 1996 [455]
	オブジェクト計算	Abadi and Cardelli, 1996 [4]
	透過型とモジュール性	Harper and Lillibridge, 1994 [196]; Leroy, 1994 [279]
	型付きアセンブリ言語	Morrisett et al., 1998 [328]

▲ 図 1-1. 計算機科学と論理学における型の歴史年表

(Haskell や Miranda のような「純粋関数的プログラミング」の意味) と構成的型理論の間の、論理学的な観点から見たつながりに着目している。証明論における関連トピックは、Goubault-Larrecq と Mackie の “*Proof Theory and Automated Deduction*” [185] で多数解説されている。論理学や哲学の中での型の歴史は、Constable [109]、Wadler [482]、Huet [229]、Pfenning [361] といった記事や、Laan の博士論文 [267]、Grattan-Guinness [186] や Sommaruga [438] といった書籍でより詳細に説明されている。

何らかのプログラミング言語が型健全であるという主張が誤っていて恥をかかないようにするには、相当量の注意深い分析が必要なことがわかる。結果として、型システムの分類、記述、研究が形式的な学問領域として浮上したのである。
—*Luca Cardelli, 1996* [82]

第2章

数学的準備

まずはじめに、共通の記法を定め、基本的な数学的事実について述べる。大半の読者はこの章を読み流し、必要に応じて見直せばよいだろう。

2.1 集合、関係、関数

定義 2.1.1. 集合に関しては標準的な記法を用いる。波括弧を用いて、集合の要素を明示的に列挙し（$\{\ldots\}$）、また、「内包表記」によって集合から別の集合を構築する方法を示す（$\{x \in S \mid \ldots\}$）。$\varnothing$ は空集合を表し、$S \setminus T$ は S と T の差集合（S の要素だが、T の要素ではない要素の集合）を表す。集合 S の大きさを $|S|$ と書く。S の冪集合（S のすべての部分集合を要素とする集合）を $\mathcal{P}(S)$ と書く。 □

定義 2.1.2. **自然数**の集合 $\{0, 1, 2, 3, 4, 5, \ldots\}$ を記号 $\mathbb{N}$ で表す。集合の要素が自然数と一対一に対応付けられるとき、その集合は**可算**であるという。 □

定義 2.1.3. 集合 $S_1, S_2, \ldots, S_n$ 上の n **項関係**とは、S_1 から S_n までの集合の各要素の組を要素とする集合の部分集合 $R \subseteq S_1 \times S_2 \times \cdots \times S_n$ である。各要素 $s_1 \in S_1$ から $s_n \in S_n$ に対し、$(s_1, \ldots, s_n)$ が R の要素であるとき、$s_1, \ldots, s_n$ は R によって**関係付け**られるという。 □

定義 2.1.4. 集合 S 上の単項関係（項が一つだけの関係）は、S 上の**述語**と呼ばれる。$s \in P$ のとき、P は要素 $s \in S$ に対して真であるという。この直観を強調するため、$s \in P$ の代わりに $P(s)$ と書くことが多い。すなわち P は、S の要素を真偽値に写す関数と見ることができる。 □

定義 2.1.5. 集合 S と T 上の**二項関係** R に対しては、$(s, t) \in R$ の代わりに $s\ R\ t$ と書くことが多い。S と T が同一の集合 U であるとき、R を U 上の二項関係と呼ぶ。 □

定義 2.1.6. 読みやすさのため、三項以上の関係は「混置」の具象構文を用いて書くことが多い。つまり、関係を表すひとまとまりの記号の列を定め、それらの記号で関係の要素を区切る。例えば、第9章の単純型付きラムダ計算の型付け関係は、$\Gamma \vdash \mathsf{s} : \mathsf{T}$ と書いて「三つ組 $(\Gamma, \mathsf{s}, \mathsf{T})$ が型付け関係に含まれる」ことを表す。 □

定義 2.1.7. 集合 S と T 上の関係 R の**定義域**（$dom(R)$ と書く）とは、$(s, t) \in R$ なる t が存在する要素 $s \in S$ からなる集合のことである。R の**値域**（$range(R)$ と書く）とは、$(s, t) \in R$ なる s が存在する要素 $t \in T$ からなる集合のことである。 □

定義 2.1.8. 集合 S と T 上の関係 R に対し、$(s, t_1) \in R$ かつ $(s, t_2) \in R$ ならば常に $t_1 = t_2$ であるとき、R は S から T への**部分関数**と呼ばれる。さらに、$dom(R) = S$ ならば、R は S から T への**全域関数**（または単に**関数**）と呼ばれる。 □

定義 2.1.9. S から T への部分関数 R は、$s \in dom(R)$ のとき、引数 $s \in S$ に対して**定義**されているといわれ、そうでないときは未定義であるといわれる。$f(x)\uparrow$ や $f(x) = \uparrow$ は「f が x に対して未定義である」ことを表し、$f(x)\downarrow$ は「f は x に対して定義されている」ことを表す。

一部の実装の章では、一部の入力に対して**失敗**しうる関数を定義する必要がある（例えば図 22-2 を見よ）。発散[†1]と、（きちんと定義され、観測可能な結果である）失敗を区別することは重要である。失敗しうる関数には、（発散もしうる）部分関数と、（結果を返すか明示的に失敗するか、常にどちらかでなければならない）全域関数のどちらもありうる。実際に、全域性を証明することに興味がある場合はよくある。f が入力 x に対して失敗の結果を返すとき、$f(x) = fail$ と書く。

形式的には、S から T への失敗しうる関数は、S から $T \cup \{fail\}$ への関数である（$fail$ は T に属さないものとする）。 □

定義 2.1.10. R を S 上の二項関係とし、P を S 上の述語とする。$s\ R\ s'$ かつ $P(s)$ のとき常に $P(s')$ であれば、P は R **によって保存される**という。 □

2.2 順序集合

定義 2.2.1. 集合 S 上の二項関係 R は、R が S のすべての要素をそれ自身に関係付けるとき、つまり、すべての $s \in S$ に対して $s\ R\ s$（すなわち $(s,s) \in R$）であるとき、**反射的**であるという。$s\ R\ t$ ならば $t\ R\ s$ であるとき、R は**対称的**であるという。$s\ R\ t$ かつ $t\ R\ u$ ならば $s\ R\ u$ であるとき、R は**推移的**であるという。$s\ R\ t$ かつ $t\ R\ s$ ならば $s = t$ であるとき、R は**反対称的**であるという。 □

定義 2.2.2. 集合 S 上の反射的推移的関係 R は、S 上の**前順序**と呼ばれる。（「前順序付き集合 S」というときは常に、S 上の特定の前順序 R を念頭に置いている。）前順序は通常 $\leq$ や $\sqsubseteq$ のような記号を用いて書かれる。$s < t$（s は t より**真に小さい**）と書くときは、$s \leq t \wedge s \neq t$ を意味する。

（S 上の）反対称的な前順序を、S 上の**半順序**という。半順序 $\leq$ は、S のすべての要素 s と t に対し $s \leq t$ か $t \leq s$ のいずれかが成り立つという性質を持つとき、**全順序**という。 □

定義 2.2.3. $\leq$ が集合 S 上の半順序であり、s と t が S の要素であるとする。要素 j が以下の性質を満たすとき、j は s と t の**結び**（または**上限**）であるという。

(1) $s \leq j$ かつ $t \leq j$
(2) 任意の要素 $k \in S$ に対し、$s \leq k$ かつ $t \leq k$ ならば、$j \leq k$

同様に、要素 $m \in S$ が以下の性質を満たすとき、m は s と t の**交わり**（または**下限**）であるという。

(1) $m \leq s$ かつ $m \leq t$
(2) 任意の要素 $n \in S$ に対し、$n \leq s$ かつ $n \leq t$ ならば、$n \leq m$ □

定義 2.2.4. 集合 S 上の反射的、推移的かつ対称的な関係を、S 上の**同値関係**という。 □

定義 2.2.5. R を集合 S 上の二項関係とする。R の**反射的閉包**とは、R を含む最小の反射的関係 R' のことである。（ここでいう「最小」とは、R 中のすべての二つ組を含む任意の反射的関係 R'' に対して、$R' \subseteq R''$ が成り立つということである。）同様に、R の**推移的閉包**とは、R を含む最小の推移的関係 R' のことである。R の推移的閉包はしばしば R^+ と書かれる。R の**反射的推移的閉包**は、R を含む最小の反射的推移的関係である。これはしばしば R^* と書かれる。 □

[†1] (訳注) ここでは未定義のこと。

演習 2.2.6. [★★ ↛]　集合 S 上の関係 R を与える。関係 R' を以下のように定義する。

$$R' = R \cup \{(s,s) \mid s \in S\}$$

つまり、R' は R 中のすべての二つ組に加え、(s,s) の形をしたすべての二つ組を含む。R' は R の反射的閉包であることを示せ。 □

演習 2.2.7. [★★↛]　関係 R の推移的閉包の、より構成的な定義を述べる。まず、二つ組の集合の列を以下のように定義する。

$$\begin{array}{lcl} R_0 & = & R \\ R_{i+1} & = & R_i \cup \{(s,u) \mid \text{ある } t \text{ に対し、} (s,t) \in R_i \text{ かつ } (t,u) \in R_i\} \end{array}$$

つまり、R_{i+1} を構成するには、R_i に対し、既に R_i に含まれている二つ組同士の「一段階の推移」で得られるすべての二つ組を加える。そして、関係 R^+ はすべての R_i の和として定義する。

$$R^+ = \bigcup_i R_i$$

R^+ が本当に R の推移的閉包であること（つまり、定義 2.2.5 で与えた条件を満たすこと）を示せ。 □

演習 2.2.8. [★★↛]　R を集合 S 上の二項関係とし、P を R によって保存される S 上の述語とする。P は R^* によっても保存されることを示せ。 □

定義 2.2.9. 集合 S 上の前順序 $\leq$ があるとする。$\leq$ における**降下列**とは、S の要素の列 $s_1, s_2, s_3, \ldots$ であって、すべての要素が直前の要素より真に小さいものをいう。すなわち、任意の i に対して $s_{i+1} < s_i$ である。（この列は有限でも無限でもよいが、次の定義に示すように、より興味があるのは無限のほうである。） □

定義 2.2.10. 前順序 $\leq$ 付きの集合 S があるとする。無限降下列が存在しないとき、$\leq$ は**整礎**であるという。例えば、自然数の通常の順序 $0 < 1 < 2 < 3 < \cdots$ は整礎であるが、整数の順序 $\cdots < -3 < -2 < -1 < 0 < 1 < 2 < 3 < \cdots$ は整礎ではない。$\leq$ に明示的に言及せず、単に S を**整礎集合**ということもある。 □

2.3 列

定義 2.3.1. **列**は要素をコンマ区切りで列挙して記述される。コンマの使い方には、要素を列の先頭または最後に追加する "cons" 操作を表す場合と、列同士の "append" 操作を表す場合がある。例えば、a を列 $3,2,1$ とし、b を列 $5,6$ としたとき、$0,a$ は列 $0,3,2,1$ を表し、$a,0$ は $3,2,1,0$ を表し、b,a は $5,6,3,2,1$ を表す。（列の列について話す必要がない限り、このようにコンマを用いても混乱は生じない。）1 から n までの数の列は $1..n$ と略記される（点は二つのみ）。列 a の長さを $|a|$ と書く。空の列は $\bullet$ か空白で書く。ある列が他の列の**並べ替え**であるとは、その二つの列が全く同じ要素からなる（順序は異なってもよい）ことをいう。 □

2.4 帰納法

帰納法による証明は、計算機科学の大半と同様、プログラミング言語理論の随所に現れる。これらの証明の多くは、以下のいずれかの原理に基づいている。

公理 2.4.1. [自然数上の通常の帰納法の原理]
P を自然数上の述語とする。このとき、

$P(0)$ であり、
かつ任意の i に対して $P(i)$ ならば $P(i+1)$
ならば、すべての n に対して $P(n)$ が成り立つ。

□

公理 2.4.2. [自然数上の完全帰納法の原理]
P を自然数上の述語とする。このとき、

各自然数 n に対して、
　任意の $i < n$ に対して $P(i)$ が成り立つとき、
　$P(n)$ が証明できる
ならば、すべての n に対して $P(n)$ が成り立つ。

□

定義 2.4.3. 自然数の二つ組上の**辞書式順序**は、「$m < m'$ か、$m = m'$ かつ $n \leq n'$ であるとき、かつそのときに限り、$(m,n) \leq (m',n')$」と定義する。 □

公理 2.4.4. [辞書式帰納法の原理]
P を自然数の二つ組上の述語とする。

各自然数の二つ組 (m,n) に対して、
　任意の $(m',n') < (m,n)$ に対して $P(m',n')$ が成り立つとき、
　$P(m,n)$ が証明できる
ならば、すべての m,n に対して $P(m,n)$ が成り立つ。

□

辞書式帰納法の原理は、帰納的証明の一部のケースが「内側の帰納法によって」進む、**多重帰納法**による証明の基本である。辞書式帰納法は、自然数の三つ組や四つ組などに一般化できる。(二つ組上の帰納法は比較的一般的である。三つ組上でもたまに有用である。四つ組以上では稀である。)

第 3 章の定理 3.3.4 では、**構造的帰納法**と呼ばれる、また別の形式の帰納法による証明を導入する。これは、項や型付けの導出といった木構造に関する証明に特に有用である。帰納的推論の数学的な基礎については第 21 章でより詳しく考える。そこでは、これら個々の帰納法原理すべてが、より深遠な一つの考え方の具体例であることがわかるだろう。

2.5 参考文献

もしこの章で要約した内容に馴染みがなければ、予備知識に関する文献を読むことから始めたほうがよいかもしれない。適切な情報源は数多くあるが、特に Winskel の著書 [496] は帰納法に関する直観を得るには良い選択だろう。Davey and Priestley [131] の冒頭部分は順序集合の素晴らしい概説になっている。Halmos [191] は基本的な集合に関する良い入門書である。

証明とは、説得術における再現可能な実験である。 *—Jim Horning*

第I部

型無しの計算体系

第 **3** 章

型無し算術式

本章で扱う体系はブール値と数の型無し計算である（30 ページの図 3-2）。対応する OCaml 実装（ウェブリポジトリでの `arith`）は第 4 章で解説する。この検査器のダウンロードやビルドに関する説明は `http://www.cis.upenn.edu/~bcpierce/tapl` にある。

型システムやその性質について厳密に述べるには、プログラミング言語のより基本的な側面を形式的に取り扱うことから始める必要がある。特に、プログラムの構文と意味論に関する表現と推論のため、明瞭・正確かつ数学的に扱いやすい道具立てが必要である。

この章と次の章では、数とブール値からなる小さな言語のために必要な道具立てを整える。この言語はほとんど考慮に値しないくらい単純だが、抽象構文、帰納的定義・証明、評価、そして実行時エラーのモデル化といった幾つかの基本的な概念を導入する単純な手段として役に立つ。第 5 章から第 7 章までは、より強力な言語である型無しラムダ計算について同じ議論を展開するが、前述の概念に加え名前束縛と代入についても扱う必要がある。さらに第 8 章では、本章の単純な言語に戻って、静的型付けの基本的な概念を導入し、正式に型システムの学習を開始する。第 9 章では静的型付けの概念をラムダ計算に拡張する。

3.1 導入

この章で使う言語は一握りの構文のみからなる。ブール定数 `true` と `false`、条件式、数値定数 0、算術演算子 `succ`（後者関数）と `pred`（前者関数）、そして 0 が与えられると `true` を返し、それ以外の数が与えられると `false` を返す判定演算子 `iszero` である。これらは次の文法でコンパクトに要約できる。

t ::=	項：
true	定数真
false	定数偽
if t then t else t	条件式
0	定数ゼロ
succ t	後者値
pred t	前者値
iszero t	ゼロ判定

この文法（および本書全体）で使用されている慣習は標準的な BNF（Aho, Sethi, and Ullman [16] 参照）に近い。最初の行（t ::=）は、これから**項**の集合を定義するということと、t という文字は項の上を動くものとして使うということを宣言している。それに続くすべての行は、それぞれ構文形式の一つの選択肢を与えている。記号 t が現れるすべての部分には、任意の項を代入できる。右端の語句は単にコメントである。

この構文の規則の右辺に現れる記号 t を**メタ変数**と呼ぶ。これは何か特定の項のためのプレースホルダーという意味で変数であり、**対象言語**（今構文を記述している単純なプログラミング言語）の変数ではなく、**メタ言語**（その記述を与えている記法）の変数である、という意味で「メタ」である。（実際には、ここでの対象言語はそもそも変数を持っていない。変数は第 5 章で導入する。）接頭辞

メタは**メタ数学**に由来している。メタ数学とは論理学の 1 分野で、数学的・論理的な推論のための体系そのもの（プログラミング言語を含む）に関する数学的な性質を主題とする。この分野は**メタ理論**という用語も与えており、ある特定の論理体系（もしくはプログラミング言語）に関してできる正しい言明の集まり、またはそのような言明の研究を意味する。それゆえ、この本で「部分型付けのメタ理論」のような言葉は「部分型付けを持つ体系の性質の形式的な考察」と理解すればよい。

本書を通じて、そのとき議論している対象言語の項を指すために、メタ変数 t やそれに近い文字の s、u、r、派生形として t_1 や s′ などを用いる。他の構文要素の式を表す文字は随時導入する。メタ変数の慣習の完全な要約は付録 B にある。

今のところ、**項**と**式**という二つの用語は同義である。**型**のような別の構文要素を持つ計算体系の議論を始める第 8 章以降では、**式**はあらゆる種類の構文的表現に対して使い（項式、型式、カインド式などを含む)、**項**は計算の構文的表現（つまりメタ変数 t に代入することのできる表現）という、より特殊な意味のためにとっておく。

この言語におけるプログラムとは、単に上述の文法で与えられた形からなる項である。以下は幾つかのプログラムの例と、それらを評価した結果である。例を簡潔にするため、数には標準的なアラビア数字を用いるが、これは形式的には succ の 0 へのネストした適用を表す。例えば succ(succ(succ(0))) は 3 と書く。

```
  if false then 0 else 1;
▸ 1

  iszero (pred (succ 0));
▸ true
```

本書を通じて、記号 ▸ は例を評価した結果を表示するために用いる。（簡単のため、結果が明らかなときや重要でないときは省略する。）組版の際、例は議論中の形式的体系に対応する実装（ここでは arith）によって自動的に処理されており、表示されている評価結果はその実装の実際の出力である。

例において、succ、pred、iszero に複合的な引数を渡すときは、読みやすさのため、括弧で囲む[†1]。項の文法は括弧については言及せず、項の**抽象構文**だけを定義している。もちろん括弧の有無は現在扱っている非常に単純な言語ではほとんど違いを生じない。通常、括弧は文法の曖昧さを解決するために用いられるが、この文法は少しの曖昧性も持たない。つまりトークン列は高々一通りの方法で項へと構文解析される。括弧と抽象構文については第 5 章（40 ページ）でもう一度議論することになる。

評価の結果は非常に簡単な形の項で、常にブール定数か数（ゼロ個以上の succ の 0 へのネストした適用）のどちらかになる。このような項は**値**と呼ばれ、項の評価順序の形式化において特別な役割を果たす。

この項の構文は succ true や if 0 then 0 else 0 のような怪しげな項の形成を許すことに注意されたい。このような項については後でもっと述べることとしよう。実際、ある意味まさにこうした例のおかげで、この小さな言語が本書の目的から見て興味深いものとなっている。これらは我々が型システムによって排除したくなる類いの、無意味なプログラムの例だからである。

[†1] 実のところ、本章の例は実際には括弧なしでも曖昧性なく構文解析できるのだが、それらを処理するために使われている実装（本書のウェブサイトでは arith と呼ぶ）は succ、pred、iszero の複合的な引数に括弧を**必要**とする。これは似たような見た目の構文を関数適用のために用いる、後の計算との一貫性のためである。

3.2 構文

ある言語の構文を定義するに際して、複数の同等な方法が存在する。既に 17 ページの文法でその一つを見た。この文法は実際には次の帰納的定義のためのコンパクトな記法にすぎない。

定義 3.2.1.［帰納的な項の定義］ 項の集合とは以下の条件を満たす最小の集合 $\mathcal{T}$ である。

(1) $\{\texttt{true}, \texttt{false}, \texttt{0}\} \subseteq \mathcal{T}$

(2) $\texttt{t}_1 \in \mathcal{T}$ ならば $\{\texttt{succ}\ \texttt{t}_1, \texttt{pred}\ \texttt{t}_1, \texttt{iszero}\ \texttt{t}_1\} \subseteq \mathcal{T}$

(3) $\texttt{t}_1 \in \mathcal{T}$ かつ $\texttt{t}_2 \in \mathcal{T}$ かつ $\texttt{t}_3 \in \mathcal{T}$ ならば $\texttt{if}\ \texttt{t}_1\ \texttt{then}\ \texttt{t}_2\ \texttt{else}\ \texttt{t}_3 \in \mathcal{T}$ □

帰納的定義はプログラミング言語の研究の至るところに現れるので、ここで少し立ち止まって、上の帰納的定義を詳しく見るだけの価値がある。一つ目の条件は、$\mathcal{T}$ に属する三つの単純な式を挙げている。二つ目と三つ目の条件は、ある種の複合的な式が $\mathcal{T}$ に属することを判断するための規則を与えている。最後に、「最小の」という言葉は $\mathcal{T}$ がこの三つの条件によって要求される要素以外を持たないということをいっている。

17 ページの文法のように、この定義は複合的な部分式に付ける括弧の使用について何もいっていない。形式的に言うと、$\mathcal{T}$ は文字列の集合ではなく、実は**木**の集合として定義されているのである。例の中での括弧の使用は、項をページ上に書くときの直線的な形と、その背後にある実際の木の形との関係を明確にする方法にすぎない。

同じ項の帰納的定義の別の略記法として、二次元の**推論規則**形式を用いる方法もある。これは論理体系を「自然演繹スタイル」で提示するためによく使われる。

定義 3.2.2.［推論規則による項の定義］ 項の集合は次の規則によって定義される。

$$\texttt{true} \in \mathcal{T} \qquad \texttt{false} \in \mathcal{T} \qquad \texttt{0} \in \mathcal{T}$$

$$\frac{\texttt{t}_1 \in \mathcal{T}}{\texttt{succ}\ \texttt{t}_1 \in \mathcal{T}} \qquad \frac{\texttt{t}_1 \in \mathcal{T}}{\texttt{pred}\ \texttt{t}_1 \in \mathcal{T}} \qquad \frac{\texttt{t}_1 \in \mathcal{T}}{\texttt{iszero}\ \texttt{t}_1 \in \mathcal{T}}$$

$$\frac{\texttt{t}_1 \in \mathcal{T} \qquad \texttt{t}_2 \in \mathcal{T} \qquad \texttt{t}_3 \in \mathcal{T}}{\texttt{if}\ \texttt{t}_1\ \texttt{then}\ \texttt{t}_2\ \texttt{else}\ \texttt{t}_3 \in \mathcal{T}}$$ □

最初の三つの規則は定義 3.2.1 の一つ目の条件を言い直したものであり、それに続く四つの規則は条件 (2) と (3) を捉えたものである。それぞれの規則は「もし線の上に列挙した前提が成立するならば、線の下の結論を導出できる」と読む。$\mathcal{T}$ がこれらの規則を満たす**最小の**集合であるという事実はしばしば（ここでの定義のように）明示的には述べられない。

用語について二点、説明すべきことがある。一点目として、（上の最初の三つの規則のように）前提のない規則は、しばしば**公理**と呼ばれる。本書では、**推論規則**という語は広い意味で使い、公理と、一つ以上の前提を持つ「規則らしい規則」との両方を含むことにする。公理はその上にくるものがないため、通常は線を描かない。二点目として、非常に細かいことを言うと、我々が「推論規則」と呼んでいるものは、前提と結論にメタ変数を含むかもしれないので、実際は**規則のスキーマ**である。形式的には、各スキーマは各メタ変数をそれに適した構文要素のすべての表現で一貫して置き換える（すなわち上の規則では、各 t をありとあらゆる項で置き換える）ことによって得られる**具体的な規則**の無限集合を表現している。

最後に、ここまでと同じ項の集合についてのもう一つの定義を、わずかに異なる、より「具体的な」スタイルで行う。これは $\mathcal{T}$ の要素を**生成**する明示的な手続きを与える。

定義 3.2.3.［具体的な項の定義］ 各自然数 i について、集合 S_i を以下のように定義する。

$$\begin{aligned} S_0 &= \emptyset \\ S_{i+1} &= \quad \{\texttt{true}, \texttt{false}, \texttt{0}\} \\ &\quad \cup \; \{\texttt{succ}\ \texttt{t}_1, \texttt{pred}\ \texttt{t}_1, \texttt{iszero}\ \texttt{t}_1 \mid \texttt{t}_1 \in S_i\} \\ &\quad \cup \; \{\texttt{if}\ \texttt{t}_1\ \texttt{then}\ \texttt{t}_2\ \texttt{else}\ \texttt{t}_3 \mid \texttt{t}_1, \texttt{t}_2, \texttt{t}_3 \in S_i\} \end{aligned}$$

そして、

$$S = \bigcup_i S_i$$

とする。S_0 は空である。S_1 は定数のみを含む。S_2 は定数に加えて、succ、pred、iszero、if のいずれかをただ一回と定数を使って構成できる語句を含む。S_3 はこれらに加えて succ、pred、iszero、if を S_2 の語句に使って構成できるすべての語句を含む。以下同様である。S はこの方法で作ることのできるすべての語句を集める。すなわち、定数のみから始まって、有限個の算術演算子と条件演算子を使って構成されるすべての語句である。□

演習 3.2.4.［★★］ S_3 が持つ要素の数を答えよ。□

演習 3.2.5.［★★］ 集合 S_i が**累積的**であること（つまり、各 i について、$S_i \subseteq S_{i+1}$ となること）を示せ。□

ここまでに見てきた複数の定義は、同じ項の集合を異なる方向から特徴付けている。定義 3.2.1 と定義 3.2.2 は単にある種の「閉包性」を満たす最小の集合として項の集合を**特徴付けている**。定義 3.2.3 は列の極限として、項の集合を実際にどう**構成する**かを示している。

議論を仕上げるために、この二つの考え方により実際に同じ集合が定義されることを検証しよう。これから、かなり詳細に証明を行い、すべての部品がどう組み合わされるかを見ていく。

命題 3.2.6. $\mathcal{T} = S$ □

［証明］ $\mathcal{T}$ はある条件を満たす最小の集合として定義された。よって、(a) S がそれらの条件を満たすことと、(b) その条件を満たす任意の集合が S を部分集合として持つこと、すなわち S が条件を満たす**最小の**集合であることを示せば十分である。

(a) について、S が定義 3.2.1 の三つの条件をすべて満たすかどうか調べる必要がある。まず最初に、$S_1 = \{\texttt{true}, \texttt{false}, \texttt{0}\}$ から、定数が S に属することは明らかである。第二に、もし $\texttt{t}_1 \in S$ ならば、($S = \bigcup_i S_i$ であるから) ある i について $\texttt{t}_1 \in S_i$ とならなければならない。しかしこのとき、S_{i+1} の定義により $\texttt{succ}\ \texttt{t}_1 \in S_{i+1}$ となり、よって $\texttt{succ}\ \texttt{t}_1 \in S$ となる。同様に、$\texttt{pred}\ \texttt{t}_1 \in S$ と $\texttt{iszero}\ \texttt{t}_1 \in S$ も成り立つ。第三に、$\texttt{t}_1 \in S$ かつ $\texttt{t}_2 \in S$ かつ $\texttt{t}_3 \in S$ ならば、同様の議論により $\texttt{if}\ \texttt{t}_1\ \texttt{then}\ \texttt{t}_2\ \texttt{else}\ \texttt{t}_3 \in S$ である。

(b) について、ある集合 S' が定義 3.2.1 の 3 条件を満たすと仮定する。i に関する完全帰納法を用いてすべての $S_i \subseteq S'$ を示す。これから $S \subseteq S'$ が得られることは明らかである。

すべての $j < i$ について $S_j \subseteq S'$ であると仮定し、その上で $S_i \subseteq S'$ を示す必要がある。S_i の定義には節が二つある（$i = 0$ の場合と $i > 0$ の場合）ので、考えるべきケースは二つある。$i = 0$ ならば $S_i = \emptyset$ であり、$\emptyset \subseteq S'$ は自明である。それ以外の場合、$i = j + 1$ となる j が存在する。$\texttt{t}$ を S_{j+1} の元であるとする。S_{j+1} はより小さい三つの集合の和集合として定義されているので、$\texttt{t}$ はそれらのうち一つに属するはずである。すなわち、三つの可能性が考えられる。(1) もし $\texttt{t}$ が定数ならば、条件 1 より $\texttt{t} \in S'$ となる。(2) もし $\texttt{t}$ がある $\texttt{t}_1 \in S_j$ に対して $\texttt{succ}\ \texttt{t}_1$、$\texttt{pred}\ \texttt{t}_1$、$\texttt{iszero}\ \texttt{t}_1$ のい

ずれかの形であるならば、帰納法の仮定より $\mathtt{t}_1 \in S'$ となり、条件 2 によって $\mathtt{t} \in S'$ が成り立つ。(3) もし t がある $\mathtt{t}_1, \mathtt{t}_2, \mathtt{t}_3 \in S_j$ に対して if $\mathtt{t}_1$ then $\mathtt{t}_2$ else $\mathtt{t}_3$ という形であるならば、やはり帰納法の仮定により、$\mathtt{t}_1$、$\mathtt{t}_2$、$\mathtt{t}_3$ はすべて S' に属し、条件 3 より t も S' に属する。

よって各 $S_i \subseteq S'$ を証明した。S がすべての S_i の和集合として定義されていることから、$S \subseteq S'$ であることが得られる。これにて証明は完了である。 □

注目すべきこととしては、この証明は馴染み深い「基底ケース / 帰納ケース」という形ではなく、自然数上の**完全**帰納法に基づいている。各 i について、i より真に小さいすべての数について所望の述語が成り立つことを仮定し、そのとき i についても成り立つことを証明した。本質的には、**すべてのステップ**は帰納ステップである。$i = 0$ のケースで特別なのは、(帰納法の仮定を利用できる) i より小さい値の集合が偶然空であるということだけである。同じことは、この本を通じて見ていく証明のほとんど (特に「構造的帰納法」による証明) についてもいえる。

3.3 項に関する帰納法

命題 3.2.6 における項の集合 $\mathcal{T}$ の明示的な特徴付けは、その元に関して推論するための、ある重要な原理を正当化する。すなわち、もし $\mathtt{t} \in \mathcal{T}$ ならば、t に関して次の三つのうちのいずれかが真でなければならない。(1) t は定数である。(2) **t より小さい**項 $\mathtt{t}_1$ が存在して、t は succ $\mathtt{t}_1$、pred $\mathtt{t}_1$、iszero $\mathtt{t}_1$ のいずれかの形である。(3) **t より小さい**項 $\mathtt{t}_1$、$\mathtt{t}_2$、$\mathtt{t}_3$ が存在して、t は if $\mathtt{t}_1$ then $\mathtt{t}_2$ else $\mathtt{t}_3$ という形である。この観察は二つの方法で活用できる。項の集合上の関数の**帰納的定義**と、項の性質の**帰納的証明**である。例えば以下は、各項 t を t で使われている定数の集合へ写す関数の単純な帰納的定義である。

定義 3.3.1. 項 t に現れる定数の集合を $Consts(\mathtt{t})$ と書き、次のように定義する。

$$
\begin{array}{lcl}
Consts(\mathtt{true}) & = & \{\mathtt{true}\} \\
Consts(\mathtt{false}) & = & \{\mathtt{false}\} \\
Consts(0) & = & \{0\} \\
Consts(\mathtt{succ}\ \mathtt{t}_1) & = & Consts(\mathtt{t}_1) \\
Consts(\mathtt{pred}\ \mathtt{t}_1) & = & Consts(\mathtt{t}_1) \\
Consts(\mathtt{iszero}\ \mathtt{t}_1) & = & Consts(\mathtt{t}_1) \\
Consts(\mathtt{if}\ \mathtt{t}_1\ \mathtt{then}\ \mathtt{t}_2\ \mathtt{else}\ \mathtt{t}_3) & = & Consts(\mathtt{t}_1) \cup Consts(\mathtt{t}_2) \cup Consts(\mathtt{t}_3)
\end{array}
$$
□

帰納的定義によって計算できる項の性質には他に、項のサイズがある。

定義 3.3.2. 項 t の**サイズ**を $size(\mathtt{t})$ と書き、次のように定義する。

$$
\begin{array}{lcl}
size(\mathtt{true}) & = & 1 \\
size(\mathtt{false}) & = & 1 \\
size(0) & = & 1 \\
size(\mathtt{succ}\ \mathtt{t}_1) & = & size(\mathtt{t}_1) + 1 \\
size(\mathtt{pred}\ \mathtt{t}_1) & = & size(\mathtt{t}_1) + 1 \\
size(\mathtt{iszero}\ \mathtt{t}_1) & = & size(\mathtt{t}_1) + 1 \\
size(\mathtt{if}\ \mathtt{t}_1\ \mathtt{then}\ \mathtt{t}_2\ \mathtt{else}\ \mathtt{t}_3) & = & size(\mathtt{t}_1) + size(\mathtt{t}_2) + size(\mathtt{t}_3) + 1
\end{array}
$$

つまり t のサイズは t の抽象構文木のノードの数である。同様に、項 t の**深さ**を $depth(\mathtt{t})$ と書き、次のように定義する。

$$
\begin{array}{lcl}
\mathit{depth}(\mathtt{true}) & = & 1 \\
\mathit{depth}(\mathtt{false}) & = & 1 \\
\mathit{depth}(\mathtt{0}) & = & 1 \\
\mathit{depth}(\mathtt{succ}\ \mathtt{t}_1) & = & \mathit{depth}(\mathtt{t}_1) + 1 \\
\mathit{depth}(\mathtt{pred}\ \mathtt{t}_1) & = & \mathit{depth}(\mathtt{t}_1) + 1 \\
\mathit{depth}(\mathtt{iszero}\ \mathtt{t}_1) & = & \mathit{depth}(\mathtt{t}_1) + 1 \\
\mathit{depth}(\mathtt{if}\ \mathtt{t}_1\ \mathtt{then}\ \mathtt{t}_2\ \mathtt{else}\ \mathtt{t}_3) & = & \max(\mathit{depth}(\mathtt{t}_1), \mathit{depth}(\mathtt{t}_2), \mathit{depth}(\mathtt{t}_3)) + 1
\end{array}
$$

言い換えれば、$\mathit{depth}(\mathtt{t})$ は定義 3.2.3 において $\mathtt{t} \in S_i$ となる最小の i である。 □

以下は、項の定数の数と項のサイズとを関係付ける単純な事実の帰納的証明である。(もちろん、この性質自体は完全に明らかである。興味深いのはその帰納的証明の形式で、これから何度も繰り返し見ることになる。)

補題 3.3.3. ある項 t の中の異なる定数の数は t のサイズ以下である (つまり $|\mathit{Consts}(\mathtt{t})| \leq \mathit{size}(\mathtt{t})$)。 □

[証明] t の深さに関する帰納法を用いる。所望の性質が t より小さいすべての項に対して成り立つと仮定し、同じ性質が t 自身でも成り立つことを証明しなければならない。考えるべきケースは三つある。

t が定数の場合

直ちに成り立つ。$|\mathit{Consts}(\mathtt{t})| = |\{\mathtt{t}\}| = 1 = \mathit{size}(\mathtt{t})$ である。

$\mathtt{t} = \mathtt{succ}\ \mathtt{t}_1$ または $\mathtt{pred}\ \mathtt{t}_1$ または $\mathtt{iszero}\ \mathtt{t}_1$ の場合

帰納法の仮定より $|\mathit{Consts}(\mathtt{t}_1)| \leq \mathit{size}(\mathtt{t}_1)$ となる。すると、$|\mathit{Consts}(\mathtt{t})| = |\mathit{Consts}(\mathtt{t}_1)| \leq \mathit{size}(\mathtt{t}_1) < \mathit{size}(\mathtt{t})$ と計算できる。

$\mathtt{t} = \mathtt{if}\ \mathtt{t}_1\ \mathtt{then}\ \mathtt{t}_2\ \mathtt{else}\ \mathtt{t}_3$ の場合

帰納法の仮定より、$|\mathit{Consts}(\mathtt{t}_1)| \leq \mathit{size}(\mathtt{t}_1)$ かつ $|\mathit{Consts}(\mathtt{t}_2)| \leq \mathit{size}(\mathtt{t}_2)$ かつ $|\mathit{Consts}(\mathtt{t}_3)| \leq \mathit{size}(\mathtt{t}_3)$ となる。ここで次のように計算する。

$$
\begin{array}{rcl}
|\mathit{Consts}(\mathtt{t})| & = & |\mathit{Consts}(\mathtt{t}_1) \cup \mathit{Consts}(\mathtt{t}_2) \cup \mathit{Consts}(\mathtt{t}_3)| \\
 & \leq & |\mathit{Consts}(\mathtt{t}_1)| + |\mathit{Consts}(\mathtt{t}_2)| + |\mathit{Consts}(\mathtt{t}_3)| \\
 & \leq & \mathit{size}(\mathtt{t}_1) + \mathit{size}(\mathtt{t}_2) + \mathit{size}(\mathtt{t}_3) \\
 & < & \mathit{size}(\mathtt{t})
\end{array}
$$

□

この証明の形式を明らかにするため、一般的な推論の原理として言及し直す。おまけとして、項に関する証明でしばしば使われる、二つのよく似た原理も併せて述べる。

定理 3.3.4. [項に関する帰納法の原理] P が項に関する述語であると仮定する。

深さに関する帰納法：

各項 s に対して、

$\mathit{depth}(\mathtt{r}) < \mathit{depth}(\mathtt{s})$ なる任意の r に対して $P(\mathtt{r})$ が成り立つとき、

$P(\mathtt{s})$ が証明できる

ならば、すべての s に対して $P(\mathtt{s})$ が成り立つ。

サイズに関する帰納法：

各項 s に対して、

$\mathit{size}(\mathtt{r}) < \mathit{size}(\mathtt{s})$ なる任意の r に対して $P(\mathtt{r})$ が成り立つとき、

$P(\mathtt{s})$ が証明できる

ならば、すべての s に対して $P(\mathtt{s})$ が成り立つ。

構造的帰納法：

各項 s に対して、

s の任意の直接の部分項 r に対して $P(\mathtt{r})$ が成り立つとき、

$P(\mathtt{s})$ が証明できる

ならば、すべての s に対して $P(\mathtt{s})$ が成り立つ。 □

[証明] 演習[★★]。 □

項の深さとサイズに関する帰納法は、自然数上の完全帰納法（公理 2.4.2）と類似している。普通の構造的帰納法は、普通の自然数の帰納法の原理（公理 2.4.1）に対応する。普通の自然数の帰納法では、帰納ステップにおいて仮定 $P(n)$ だけから $P(n+1)$ を証明する必要がある。

自然数の帰納法にあるさまざまなスタイルと同じように、項に関する帰納法のどの原理を選ぶかについては、目の前の証明がより簡単な構造となるように決める。形式的には、それらは相互に導出可能である。単純な証明においては、サイズに関する帰納法と、深さに関する帰納法と、構造的帰納法のいずれによって議論しても、一般的に違いはほとんどない。スタイルの問題としては、慣習的に、なるべく構造的帰納法を用いる。なぜなら、項の上で直接議論でき、数を経由するような遠回りの必要がないためである。

項に関する帰納法による証明のほとんどは似た構造を持つ。帰納法の各ステップにおいては、項 t が与えられ、ある性質 P が t のすべての部分項（あるいは、より小さな項すべて）に対して成り立つと仮定した上で、t に対して P が成り立つことを示す。そのために、t の取りうる形（true、false、条件式、0 など）のそれぞれを別々に考え、それぞれの場合で、その形を取る任意の t に対して P が必ず成り立つことを論じる。この構造の中で、個別の帰納的証明ごとに変化する部分は個々のケースの議論の詳細だけであるから、慣習的に、変化しない部分を省いて次のように証明を書く。

[証明] t 上の帰納法による。

t = true の場合

... $P(\mathtt{true})$ を示す ...

t = false の場合

... $P(\mathtt{false})$ を示す ...

t = if $\mathtt{t}_1$ then $\mathtt{t}_2$ else $\mathtt{t}_3$ の場合

... $P(\mathtt{t}_1)$、$P(\mathtt{t}_2)$、$P(\mathtt{t}_3)$ を用いて $P(\mathtt{if}\ \mathtt{t}_1\ \mathtt{then}\ \mathtt{t}_2\ \mathtt{else}\ \mathtt{t}_3)$ を示す ...

(他の構文形式についても同様である。) □

多くの帰納的な議論（補題 3.3.3 の証明も含む）においては、上の書き方ですら細かすぎて無駄である。多くの場合、基底ケース（項 t が部分項を持たない場合）においては $P(\mathtt{t})$ が直ちに成り立ち、帰納ケースにおいては、帰納法の仮定を t の部分項へ適用し、その結果を何らかの全く自明な方法で組み合わせれば $P(\mathtt{t})$ が得られる。実のところ、読み手にとっては、単に証明を（帰納法の仮定を念頭に置きながら構文を調べて）その場で再生成するほうが、書き出された証明を確認するより**簡単**である。そのような場合には、単に「t 上の帰納法による」と書くだけで、何ら問題のない証明となる。

3.4　意味論のスタイル

言語の構文は厳密に定式化したので、次は項がどう評価されるか（つまり言語の**意味論**）について同様に正確な定義が必要になる。意味論の形式化には三つの基本的なアプローチがある。

(1) **操作的意味論**は、簡単な**抽象機械**を定義することで、プログラミング言語の振る舞いを規定する。この機械は低水準のマイクロプロセッサの命令セットではなく、項を機械語として扱うという意味で「抽象的」である。単純な言語では、機械の**状態**は単なる項であり、機械の振る舞いは**遷移関数**で定義される。この関数は、各状態について、項の単純化ステップの実行により次の状態を与えるか、もしくは機械が停止したことを宣言する。項 t の**意味**は、t を初期状態として動き始めた機械が到達する最終状態と考えられる[†2]。

一つの言語に対して二つ以上の異なる操作的意味論を与えることが役に立つことがある。つまり、より抽象的で、プログラマが書くような項に似た機械状態を使う意味論の他に、その言語の実際のインタプリタやコンパイラが操作する構造に近いものを使う、別の意味論を与えるのである。これらの異なる意味論で与えられた機械の振る舞いが、同じプログラムを実行したとき何かしらの適切な意味で一致することを示せれば、結果としてその言語の実装の正しさを証明したことになる。

(2) **表示的意味論**は、意味をより抽象的に見る。項の意味は単なる機械状態の列ではなく、数値や関数などの数学的な対象として捉えられる。言語に表示的意味を与えることは、**意味領域**の集まりを見つけ、項を意味領域の元に写す**解釈関数**を定義することである。さまざまな言語機能をモデル化するための適切な意味領域を探すことは、**領域理論**と呼ばれる、豊かで洗練された研究領域になっている。

表示的意味論の主な長所の一つは、評価の生々しい詳細を取り去り、言語の本質的概念を浮き彫りにすることである。選ばれた意味領域の集まりの性質を用いて、プログラムの振る舞いに関して推論するための強力な法則を導くことができる。例えば、二つのプログラムが厳密に同じ振る舞いをすることを証明するための法則や、プログラムの振る舞いが何らかの仕様を満たすことを証明するための法則である。最後に、多くの場合、ある選ばれた意味領域の集まりの性質から、ある言語においてさまざまなことが（望ましいかそうでないかにかかわらず）起こりえないということが直ちに明らかになる。

(3) **公理的意味論**は、これらの法則に対するより直接的なアプローチである。最初にプログラムの振る舞いを（操作的または表示的意味を与えることによって）定義し、この定義から法則を導き出すというのではなく、公理的意味論は言語の定義として法則**自身**を取る。項の意味はまさにその項について証明できる事実である。

公理的意味論の美しさは、プログラムに関する推論のプロセスに専心していることである。この考え方が、**不変条件**といった強力なアイディアを計算機科学にもたらした。

1960 年代から 1970 年代にかけて、一般的に操作的意味論は他のスタイルよりも劣っているとみなされていた。つまり、言語機能を間に合わせで定義するには役に立つが、エレガントでなく数学的にも弱いと考えられていた。しかし、1980 年代において抽象的な方法が段々と困難な技術的問題にぶつかってゆく一方で[†3]、それと比較して操作的意味論の単純さと柔軟さは、ますますその魅力を増していった。特に、この分野において多くの研究者が新しい進展をもたらしたことが背景にあ

[†2] 厳密に言えば、ここで記述しているのは操作的意味論の**小ステップ**スタイルである（**構造的操作的意味論**（Plotkin [387]）と呼ばれることもある）。演習 3.5.17 では別の**大ステップ**スタイルを導入する（**自然意味論**（Kahn [247]）と呼ばれることもある）。この意味論では、抽象機械の一回の遷移が項を最終的な結果に評価する。

[†3] 表示的意味論にとっての**鬼門**は非決定性と並行性の扱いであると判明した。公理的意味論の鬼門は手続きだった。

𝔹（型無し）

構文		評価	$\boxed{t \longrightarrow t'}$
t ::=	項：	if true then t_2 else t_3 ⟶ t_2	(E-IfTrue)
true	定数真	if false then t_2 else t_3 ⟶ t_3	(E-IfFalse)
false	定数偽	$\dfrac{t_1 \longrightarrow t_1'}{\texttt{if } t_1 \texttt{ then } t_2 \texttt{ else } t_3 \longrightarrow \texttt{if } t_1' \texttt{ then } t_2 \texttt{ else } t_3}$	(E-If)
if t then t else t	条件式		
v ::=	値：		
true	真		
false	偽		

▲図 3-1.　ブール値（**B**）

る。その進展としては、Plotkin による構造的操作的意味論 [387] に始まり、Kahn による自然意味論 [247]、Milner の CCS 上の研究 [308; 309; 311] などが挙げられる。彼らはよりエレガントな形式化を導入し、表示的意味論の文脈で開発された強力な数学的技法をどの程度、操作的な枠組みへ持ち込めるかを示した。操作的意味論はそれ自体で精力的な研究分野になっており、プログラミング言語の定義やそれらの性質の研究のためによく選ばれる手法である。この本では専ら操作的意味論を用いる。

3.5　評価

数はいったんわきに置いて、ブール式だけの操作的意味論から始める。図 3-1 はその定義を要約したものである。これからその細部を見ていこう。

図 3-1 の左側は二つの式の集合を定義する文法である。一つ目は項の構文の再掲である（利便性のため）。二つ目は**値**と呼ばれる、評価の最終結果となりうる項の部分集合を定義する。ここでは値は定数 true と false だけである。メタ変数 v はこの本では専ら値のために使われる。

右側は項の**評価関係**[†4]を定義する。評価関係は $t \longrightarrow t'$ と書かれ、「t が 1 ステップで t′ に評価される」と読む。直観的には、ある時点における抽象機械の状態が t であるならば、機械は計算を 1 ステップ進めて状態を t′ へ変えることができる、ということである。この関係は三つの推論規則（最初の二つは前提を持たないため、二つの公理と一つの規則といってもよい）によって定義される。

一つ目の規則 E-IfTrue の意味は、評価の対象である項が、条件部が定数 true そのものであるような条件式であるとき、機械は新しい状態（つまり次に評価される項）として then 節の t_2 を残し、条件式を捨てるということである。同様に E-IfFalse は、条件部がそのまま false であるような条件式を 1 ステップで else 節の t_3 に評価する。これらの規則名の E- は評価関係の一部であるということを表している。他の関係の規則には違う接頭辞を付ける。

三つ目の評価規則 E-If はより興味深い。これは、条件部 t_1 が t_1' に評価されるとき、条件式全体 if t_1 then t_2 else t_3 を if t_1' then t_2 else t_3 へ評価するといっている。抽象機械の観点では、状態が単に t_1 である**他の**機械が 1 ステップで状態 t_1' に進めるとき、状態 if t_1 then t_2 else t_3

[†4] この関係には**簡約**という用語を選ぶ専門家もいる。その場合**評価**は演習 3.5.17 で述べるような、項を直接最後の値に写す「大ステップ」のために用いられる。

を持つ機械は状態 $\texttt{if } t_1' \texttt{ then } t_2 \texttt{ else } t_3$ へ進める。

これらの規則が何を言って**いないか**ということは、何を言っているかということと同じくらい重要である。定数 `true` と `false` はどの規則の左側にも現れないので、それ以上評価されることはない。また、`if` 全体が評価される前に `then` 節や `else` 節の部分式が評価される規則もない。例えば、項

```
if true then (if false then false else false) else true
```

は `if true then false else true` には評価されない。ただ一つの選択肢は、E-IfTrue を用いて外側の条件式を先に評価することである。規則間のこの相互作用は、条件式に対する特定の**評価戦略**を定めている（これは一般的なプログラミング言語における、馴染み深い評価順序に対応する）。条件式を評価するには、まず条件部を評価する必要がある。もし条件部自身が条件式ならば、さらに**それの**条件部を先に評価する必要がある。以下同様である。規則 E-IfTrue と E-IfFalse はこのプロセスが終わり、条件部が完全に評価された条件式を得たときにどうすべきかを述べている。ある意味では、E-IfTrue と E-IfFalse が本当の評価の仕事をし、E-If はどこでその仕事をすべきかを決める手助けをしている。規則におけるこれらの異なる特性は、E-IfTrue と E-IfFalse を**計算規則**と呼び、E-If を**合同規則**と呼ぶことで強調される場合がある。

これらの直観をもう少し正確にするために、評価関係を次のように形式的に定義できる。

定義 3.5.1. 推論規則の**インスタンス**とは、規則の結論や（前提があれば）前提のメタ変数それぞれに対し、一貫して同じ項による置き換えを行ったものである。 □

例えば、

$$\texttt{if true then true else (if false then false else false)} \longrightarrow \texttt{true}$$

は E-IfTrue のインスタンスであって、規則に現れる二つの t_2 を `true` に置き換え、t_3 を `if false then false else false` に置き換えたものである。

定義 3.5.2. 規則がある関係によって**満たされる**とは、規則の任意のインスタンスについて、結論がその関係に属するか、または前提のうちの一つが属さないことである。 □

定義 3.5.3. **1 ステップ評価**関係 $\longrightarrow$ とは、図 3-1 の三つの規則を満たす、項に関する最小の二項関係である。(t, t') がこの評価関係の元であるとき、「**評価関係式**（もしくは**評価判断式**）$t \longrightarrow t'$ は**導出可能**である」という。 □

ここに「最小」という語を置いたことで、評価関係式 $t \longrightarrow t'$ が導出可能であるとき、かつそのときに限り、その関係式が規則によって正当化されることになる。すなわち評価関係式 $t \longrightarrow t'$ は、公理 E-IfTrue、E-IfFalse のどちらかのインスタンスであるか、または前提が導出可能である規則 E-If のインスタンスの結論である。与えられた評価関係式が導出可能であることを証明するには、葉が E-IfTrue か E-IfFalse のインスタンスであり、内部ノードのラベルが E-If のインスタンスであるような**導出木**を示せばよい。例えば、ページからはみ出ないように

$$\begin{aligned} s &\overset{\text{def}}{=} \texttt{if true then false else false} \\ t &\overset{\text{def}}{=} \texttt{if s then true else true} \\ u &\overset{\text{def}}{=} \texttt{if false then true else true} \end{aligned}$$

と略記したとき、次の評価関係式

$$\texttt{if t then false else false} \longrightarrow \texttt{if u then false else false}$$

の導出可能性は以下の導出木によって示される。

$$\dfrac{\dfrac{\dfrac{}{\texttt{s} \longrightarrow \texttt{false}}\ \text{E-IfTrue}}{\texttt{t} \longrightarrow \texttt{u}}\ \text{E-If}}{\texttt{if t then false else false} \longrightarrow \texttt{if u then false else false}}\ \text{E-If}$$

この構造は枝分かれを全く持たないので、木と呼ぶのは少しおかしいと思うかもしれない。実のところ、評価関係式を示す導出木はいつもこのような細い形をすることになる。複数の前提を持つ評価規則はないので、導出木が分岐することはありえないのである。帰納的に定義された他の関係で、幾つかの規則が複数の前提を持つようなもの（例えば型付け）を考えるときに、この用語はより納得がいくものになるだろう。

評価関係式 $\texttt{t} \longrightarrow \texttt{t}'$ が導出可能であることと、$\texttt{t} \longrightarrow \texttt{t}'$ を根に持つ導出木が存在することが同値であるという事実は、評価関係の性質を推論するときにしばしば役立つ。特にこの事実から、**導出に関する帰納法**と呼ばれる証明技法を直接導ける。次の定理の証明でこの技法を説明する。

定理 3.5.4.［1 ステップ評価の決定性］ $\texttt{t} \longrightarrow \texttt{t}'$ かつ $\texttt{t} \longrightarrow \texttt{t}''$ ならば $\texttt{t}' = \texttt{t}''$ となる。 □

［証明］ $\texttt{t} \longrightarrow \texttt{t}'$ の導出に関する帰納法を使う。帰納法の各ステップで、すべてのより小さい導出に関して所望の結果が成り立つと仮定し、導出の根で用いられた規則に対する場合分けによって証明を進める。(ここでの帰納法は評価列の長さに関する帰納法ではないことに注意されたい。1 ステップの評価のみに着目している。「評価導出」の構造は簡約される項の構造に直接従っているので、単に $\texttt{t}$ の構造に関する帰納法を行っているともいえる。代わりに、$\texttt{t} \longrightarrow \texttt{t}''$ の導出に関する帰納法を使ってもかまわない。)

もし、$\texttt{t} \longrightarrow \texttt{t}'$ の導出で最後に使った規則が E-IfTrue だったとすると、$\texttt{t}$ は $\texttt{if}\ \texttt{t}_1\ \texttt{then}\ \texttt{t}_2\ \texttt{else}\ \texttt{t}_3$ という形をしており、$\texttt{t}_1 = \texttt{true}$ であることがわかる。しかし、$\texttt{t}_1 = \texttt{true}$ と $\texttt{t}_1 = \texttt{false}$ の両方を満たすことはできないので、$\texttt{t} \longrightarrow \texttt{t}''$ の導出の最後の規則が E-IfFalse ではないことは明らかである。さらにこの導出の最後の規則は E-If にもなりえない。なぜなら、E-If 規則は何らかの $\texttt{t}_1'$ についての $\texttt{t}_1 \longrightarrow \texttt{t}_1'$ という前提を必要とするが、既に述べたように $\texttt{true}$ は何にも評価されないからである。つまり二つ目の導出 $\texttt{t} \longrightarrow \texttt{t}''$ の最後の規則になりうるのは E-IfTrue だけであり、すなわち $\texttt{t}' = \texttt{t}''$ が直ちに成り立つ。

同じように、$\texttt{t} \longrightarrow \texttt{t}'$ の導出の最後の規則が E-IfFalse だった場合、$\texttt{t} \longrightarrow \texttt{t}''$ の導出の最後の規則は同じである必要があり、所望の結果は直ちに成立する。

残るは、$\texttt{t} \longrightarrow \texttt{t}'$ の導出で最後に使われた規則が E-If だった場合、この規則の形より、$\texttt{t}$ は $\texttt{if}\ \texttt{t}_1\ \texttt{then}\ \texttt{t}_2\ \texttt{else}\ \texttt{t}_3$ という形であり、また何らかの $\texttt{t}_1'$ について $\texttt{t}_1 \longrightarrow \texttt{t}_1'$ が成り立つ。上での推論と同様に $\texttt{t} \longrightarrow \texttt{t}''$ の導出の最後の規則は E-If である必要があり、よって、$\texttt{t}$ は（既にわかっている通り）$\texttt{if}\ \texttt{t}_1\ \texttt{then}\ \texttt{t}_2\ \texttt{else}\ \texttt{t}_3$ で、かつ、ある $\texttt{t}_1''$ が存在して $\texttt{t}_1 \longrightarrow \texttt{t}_1''$ が成り立つことになる。ここで、(導出 $\texttt{t}_1 \longrightarrow \texttt{t}_1'$ と $\texttt{t}_1 \longrightarrow \texttt{t}_1''$ は、それぞれ元の導出 $\texttt{t} \longrightarrow \texttt{t}'$ と $\texttt{t} \longrightarrow \texttt{t}''$ の部分導出であるため）帰納法の仮定を適用し、$\texttt{t}_1' = \texttt{t}_1''$ を得る。よって、$\texttt{t}' = \texttt{if}\ \texttt{t}_1'\ \texttt{then}\ \texttt{t}_2\ \texttt{else}\ \texttt{t}_3 = \texttt{if}\ \texttt{t}_1''\ \texttt{then}\ \texttt{t}_2\ \texttt{else}\ \texttt{t}_3 = \texttt{t}''$ となり、これは所要の通りである。 □

演習 3.5.5.［★］ 上の証明で使われている帰納法の原理を定理 3.3.4 のスタイルで抜き出せ。 □

1 ステップ評価関係は、与えられた項の評価において、抽象機械がある状態から次の状態へどう移行するかを示している。しかし我々はプログラマとして、評価の最終結果（すなわち、機械がそれ以上ステップを**進められない**状態）にも興味がある。

定義 3.5.6. 項 t が**正規形**であるとは、当てはまる評価規則がない、つまり $t \longrightarrow t'$ となる t' が存在しないということである。（正規形という語は、「正規形の項 t」のように項の形を指す場合と、「正規形 t」のように項そのものを指す場合との両方に用いる。）　□

既に見たように、現在の体系において true と false は正規形である（なぜなら、すべての評価規則の左側は最も外側に if 構築子を持ち、true または false が左側にあるようなインスタンスを作ることは、明らかに不可能だからである）。このことを値に関する事実として一般化すると、次のように述べられる。

定理 3.5.7. すべての値は正規形である。　□

体系を改良して算術式を入れるときも（また、今後の章で他の構文を追加するときも）、この定理は有効なままであるようにする。正規形であることは、値である（完全に評価された結果である）ということ**そのもの**の一部である。このことが成り立たない言語定義はいずれも、壊れているという他にない。

この体系では定理 3.5.7 の逆も成り立つ。つまり、すべての正規形は値である。このことは一般には成り立たない。実のところ、値ではない正規形は、**実行時エラー**を解析する際に極めて重要な役割を果たす。このことは、この章の後のほうで算術式を導入するときにわかるだろう。

定理 3.5.8. t が正規形であるならば、t は値である。　□

[証明]　t は値ではないと仮定する。t が正規形でないことは、t に関する構造的帰納法によって簡単に示せる。

t は値ではないので、ある t_1、t_2、t_3 を使った if t_1 then t_2 else t_3 の形をしているはずである。ここで t_1 の取りうる形を考える。

もし t_1 = true ならば明らかに t は正規形ではない。なぜなら、E-IfTrue の左側に適合するからである。t_1 = false の場合も同様である。

もし t_1 が true でも false でもないなら、t_1 は値ではない。帰納法の仮定を適用して、t_1 が正規形でないことを得る。つまり、ある t_1' が存在して、$t_1 \longrightarrow t_1'$ を満たす。しかし、このことは E-If 規則を使って $t \longrightarrow$ if t_1' then t_2 else t_3 を導けることを意味する。つまり、この場合も t は正規形ではない。　□

評価の複数ステップを一つの大きな状態遷移として見られると便利なことがある。このために、ある項と、そこからゼロ個以上の 1 ステップ評価によって導出できるすべての項とを関連付ける、多ステップ評価の関係を定義する。

定義 3.5.9. **多ステップ評価**関係 $\longrightarrow^*$ とは 1 ステップ評価の反射的推移的閉包である。つまり、(1) $t \longrightarrow t'$ ならば $t \longrightarrow^* t'$、(2) すべての t について $t \longrightarrow^* t$、(3) $t \longrightarrow^* t'$ かつ $t' \longrightarrow^* t''$ ならば $t \longrightarrow^* t''$、の 3 条件を満たす最小の関係である。　□

演習 3.5.10. [★]　定義 3.5.9 を推論規則の形で再定義せよ。　□

多ステップ評価の明示的な記法があることで、例えば次のような事実を述べることが簡単になる。

定理 3.5.11.［正規形の一意性］ $\mathtt{t} \longrightarrow^* \mathtt{u}$ かつ $\mathtt{t} \longrightarrow^* \mathtt{u}'$ で、$\mathtt{u}$ と $\mathtt{u}'$ が両方正規形ならば、$\mathtt{u} = \mathtt{u}'$ が成り立つ。 □

［証明］ 1 ステップ評価の決定性（定理 3.5.4）より成り立つ。 □

算術式に進む前に最後に考える評価の性質は、**すべて**の項が値に評価できるという事実である。これもまた明らかに、再帰関数定義のような機能を持つ、より強力な言語では必ずしも成り立たない性質である。成り立つ場合でも、その証明は一般に、ここで見るものよりもずっと巧妙になる。第 12 章ではこの点について振り返り、ある種の言語に対する停止性の証明を支えるものとして型システムを用いる方法を示す。

計算機科学では、ほとんどの停止性の証明は次のような同じ基本形を持つ[†5]。まず、整礎集合 S を選び、「機械状態」（ここでは項）を S の中へ写す関数 f を与える。次に、機械状態 $\mathtt{t}$ が次の状態 $\mathtt{t}'$ へと進めるならば $f(\mathtt{t}') < f(\mathtt{t})$ となることを示す。$\mathtt{t}$ から始まる評価ステップの無限列は、f によって、S の要素の無限降下列に写せることがわかる。S は整礎であるので、無限の降下列は存在しえない。したがって、無限評価列は存在しない。関数 f はしばしば評価関係における**停止尺度**と呼ばれる。

定理 3.5.12.［評価の停止性］ すべての項 $\mathtt{t}$ に対して、ある正規形 $\mathtt{t}'$ が存在し、$\mathtt{t} \longrightarrow^* \mathtt{t}'$ を満たす。 □

［証明］ 次の観察による。各評価ステップは項のサイズを減らす。自然数上の通常の順序は整礎であるので、項のサイズが停止尺度となる。 □

演習 3.5.13.［推奨, ★★］

(1) 図 3-1 に次の規則を追加する。

$$\mathtt{if\ true\ then\ t_2\ else\ t_3} \longrightarrow \mathtt{t_3} \qquad \text{(E-FUNNY1)}$$

このとき、上で議論した定理 3.5.4、定理 3.5.7、定理 3.5.8、定理 3.5.11、定理 3.5.12 のうち成り立つままであるものはどれか。

(2) 次の規則を代わりに追加する。

$$\frac{\mathtt{t_2} \longrightarrow \mathtt{t_2'}}{\mathtt{if\ t_1\ then\ t_2\ else\ t_3} \longrightarrow \mathtt{if\ t_1\ then\ t_2'\ else\ t_3}} \qquad \text{(E-FUNNY2)}$$

このとき、上の定理のうち成り立つままのものはどれか。それぞれの証明は変更する必要があるか。 □

次にやるべきことは、評価の定義を算術式に拡張することである。図 3-2 に定義の新しい部分を要約する（図 3-2 の右上の表記は、この図が図 3-1 の拡張であり、独立した言語ではないことを表す）。

今回も、項の定義は 3.1 節で見たものの繰り返しである。値の定義に、**数値**を表す新しい構文要素の導入が必要となっている点で少し興味深い。直観的には、算術式を評価した最終結果は数であ

[†5] 第 12 章では、これよりもやや複雑な構造を持つ停止性の証明を見ることになる。

𝔹 ℕ（型無し）　　　　　　　　　　　　　　　　　　　　**B**（図 3-1）の拡張

新しい構文形式

t ::=	...	項：
	0	定数ゼロ
	succ t	後者値
	pred t	前者値
	iszero t	ゼロ判定
v ::=	...	値：
	nv	数値
nv ::=		数値：
	0	ゼロ
	succ nv	後者値

新しい評価規則　　$\boxed{t \longrightarrow t'}$

$$\frac{t_1 \longrightarrow t_1'}{\text{succ } t_1 \longrightarrow \text{succ } t_1'} \quad \text{(E-SUCC)}$$

$$\text{pred } 0 \longrightarrow 0 \quad \text{(E-PREDZERO)}$$

$$\text{pred (succ } nv_1) \longrightarrow nv_1 \quad \text{(E-PREDSUCC)}$$

$$\frac{t_1 \longrightarrow t_1'}{\text{pred } t_1 \longrightarrow \text{pred } t_1'} \quad \text{(E-PRED)}$$

$$\text{iszero } 0 \longrightarrow \text{true} \quad \text{(E-ISZEROZERO)}$$

$$\text{iszero (succ } nv_1) \longrightarrow \text{false} \quad \text{(E-ISZEROSUCC)}$$

$$\frac{t_1 \longrightarrow t_1'}{\text{iszero } t_1 \longrightarrow \text{iszero } t_1'} \quad \text{(E-ISZERO)}$$

▲ 図 3-2.　算術式（**NB**）

り、それは 0 か、またはある数に後者関数を適用したもののどちらかである。（ただし任意の値に後者関数を適用したものではない。succ(true) は値ではなくエラーとしたい。）

図 3-2 の右側にある評価規則は図 3-1 と同じパターンになっている。四つの計算規則（E-PREDZERO、E-PREDSUCC、E-ISZEROZERO、E-ISZEROSUCC）は、演算 pred と iszero が数に対して適用されたときにどのように振る舞うかを示しており、三つの合同規則（E-SUCC、E-PRED、E-ISZERO）は複合的な項において、評価を「最初の」部分項に対して行わせる。

厳密には、定義 3.5.3 を復唱すべきである（「算術式上の 1 ステップ評価関係は次の図 3-1 と図 3-2 の規則のすべてのインスタンスを満たす最小の関係である ...」）。このような決まり文句によるスペースの無駄を避けるため、「すべてのインスタンスを満たす最小の関係 ...」は言外の前提として、推論規則はそれのみで関係の定義を構成する、と考えるのが慣習となっている。

数値の構文要素（nv）はこの規則で重要な役割を果たす。例えば、E-PREDSUCC 規則で左側が（pred (succ t_1) などではなく）pred (succ nv_1) となっていることは、この規則を用いても pred (succ (pred 0)) を pred 0 に評価できないことを意味する。それをするためにはメタ変数 nv_1 を pred 0 で具体化する必要があるが、pred 0 は数値ではないからである。その代わり、項 pred (succ (pred 0)) の評価における次のステップは、次の導出木で一意に与えられる。

$$\cfrac{\cfrac{\cfrac{}{\text{pred } 0 \longrightarrow 0}\ \text{E-PREDZERO}}{\text{succ (pred 0)} \longrightarrow \text{succ } 0}\ \text{E-SUCC}}{\text{pred (succ (pred 0))} \longrightarrow \text{pred (succ 0)}}\ \text{E-PRED}$$

演習 3.5.14. [★★]　定理 3.5.4 が算術演算の評価関係に関しても成り立つことを示せ。すなわち、$t \longrightarrow t'$ かつ $t \longrightarrow t''$ ならば $t' = t''$ であることを示せ。　□

言語の操作的意味論を形式化すると、**すべての項**（ここでの pred 0 や succ false のような項も含む）の振る舞いを規定することになる。図 3-2 の規則では、0 の前者値は 0 と定義してある。一方、false へ後者関数を適用した評価結果は一切定義されていない（つまり正規形である）。このような項を**行き詰まり状態**であるという。

定義 3.5.15. 項が正規形であるが値ではないとき、その項は**行き詰まり状態**であるという。 □

「行き詰まり状態」によって、我々の簡単な機械における**実行時エラー**を単純に表すことができる。直観的には、プログラムが「無意味な状態」に辿り着いたので、操作的意味論が何をすべきかわからない、という状況を特徴付けている。言語のより具体的な実装では、このような状態は（セグメンテーションフォルト、不正な命令の実行など）計算機で起こったさまざまな種類の不具合に対応する。ここでは、この種のまずい振る舞いをすべて「行き詰まり状態」という単一の概念にまとめる。

演習 3.5.16. [推奨, ★★★] 抽象機械の無意味な状態を形式化する別の方法として、wrong と呼ばれる新しい項を導入し、今の意味論では行き詰まり状態になってしまう状況すべてで wrong を明示的に生成する規則を加え操作的意味論を拡張するという方法がある。これを詳細に行うには、次の二つの新しい構文要素を導入する。

badnat ::=	数でない正規形：
wrong	実行時エラー
true	定数真
false	定数偽
badbool ::=	ブール値でない正規形：
wrong	実行時エラー
nv	数値

そして、以下の規則で評価関係を拡張する。

$$\texttt{if}\ \text{badbool}\ \texttt{then}\ \texttt{t}_1\ \texttt{else}\ \texttt{t}_2 \longrightarrow \texttt{wrong} \qquad \text{(E-IF-WRONG)}$$

$$\texttt{succ}\ \text{badnat} \longrightarrow \texttt{wrong} \qquad \text{(E-SUCC-WRONG)}$$

$$\texttt{pred}\ \text{badnat} \longrightarrow \texttt{wrong} \qquad \text{(E-PRED-WRONG)}$$

$$\texttt{iszero}\ \text{badnat} \longrightarrow \texttt{wrong} \qquad \text{(E-ISZERO-WRONG)}$$

実行時エラーの二つの取り扱いが一致することを、(1)「二つの取り扱いが一致する」という直観を正確に述べる方法を見つけ、(2) 証明せよ。プログラミング言語に関する証明をするときによくあることだが、ここで難しいのは証明すべきことの正確な定式化であり、証明自身は単純になるはずである。 □

演習 3.5.17. [推奨, ★★★] 操作的意味論にはよく用いられる二つのスタイルがある。この本で用いるものは**小ステップ**スタイルと呼ばれる。これは、評価関係の定義が、個々の計算のステップの繰り返しによって項が値になるまでどのように書き換えられるかを、1 ステップごとに示すからである。その上で多ステップ評価関係を定義して、（多数回のステップで）項が値に評価されることに

ついて議論できる。**大ステップ**意味論（または**自然意味論**）と呼ばれるもう一つのスタイルは、「この項はこの最終的な値に評価される」という概念を直接定式化し、$\mathtt{t} \Downarrow \mathtt{v}$ と書く。本章のブール式と算術式の言語における大ステップ評価規則は次のようになる。

$$\mathtt{v} \Downarrow \mathtt{v} \tag{B-Value}$$

$$\frac{\mathtt{t_1} \Downarrow \mathtt{true} \qquad \mathtt{t_2} \Downarrow \mathtt{v_2}}{\mathtt{if\ t_1\ then\ t_2\ else\ t_3} \Downarrow \mathtt{v_2}} \tag{B-IfTrue}$$

$$\frac{\mathtt{t_1} \Downarrow \mathtt{false} \qquad \mathtt{t_3} \Downarrow \mathtt{v_3}}{\mathtt{if\ t_1\ then\ t_2\ else\ t_3} \Downarrow \mathtt{v_3}} \tag{B-IfFalse}$$

$$\frac{\mathtt{t_1} \Downarrow \mathtt{nv_1}}{\mathtt{succ\ t_1} \Downarrow \mathtt{succ\ nv_1}} \tag{B-Succ}$$

$$\frac{\mathtt{t_1} \Downarrow \mathtt{0}}{\mathtt{pred\ t_1} \Downarrow \mathtt{0}} \tag{B-PredZero}$$

$$\frac{\mathtt{t_1} \Downarrow \mathtt{succ\ nv_1}}{\mathtt{pred\ t_1} \Downarrow \mathtt{nv_1}} \tag{B-PredSucc}$$

$$\frac{\mathtt{t_1} \Downarrow \mathtt{0}}{\mathtt{iszero\ t_1} \Downarrow \mathtt{true}} \tag{B-IszeroZero}$$

$$\frac{\mathtt{t_1} \Downarrow \mathtt{succ\ nv_1}}{\mathtt{iszero\ t_1} \Downarrow \mathtt{false}} \tag{B-IszeroSucc}$$

この言語において小ステップと大ステップの意味論が一致すること、つまり $\mathtt{t} \longrightarrow^* \mathtt{v}$ のとき、かつそのときに限り $\mathtt{t} \Downarrow \mathtt{v}$ が成り立つことを示せ。 □

演習 3.5.18. [★★ ↛] 評価戦略を変えて、`if` 式の `then` 節と `else` 節が（この順序で）条件部より**先に**評価されるようにしたいとする。この効果を達成するために評価規則をどう変えればよいか示せ。 □

3.6 注記

抽象構文や具象構文、構文解析などについてのアイディアは、何十冊ものコンパイラの教科書で解説されている。帰納的定義、推論規則の体系、帰納法による証明は Winskel [496] や Hennessy [205] でより詳細に取り扱われている。

本章で用いた操作的意味論のスタイルは Plotkin による技術報告 [387] まで遡ることができる。大ステップスタイル（演習 3.5.17）は Kahn [247] によって開発された。より詳細な内容が知りたい場合は Astesiano [28] と Hennessy [205] を見よ。

構造的帰納法は Burstall [69] によって計算機科学に導入された。

Q： なぜわざわざプログラミング言語に関する証明をするのか？ 定義が正しければ、証明はたいてい退屈なのに。

A： 定義はたいてい間違っている。

—出典不詳

第 4 章

算術式の ML 実装

この章のコードはウェブ上のリポジトリ http://www.cis.upenn.edu/~bcpierce/tapl の arith 実装にある。実装のダウンロードおよびビルドの方法もそこにある。

前章で見たような形式的定義は、定義の背後にある直観が具体的な実装に結びつき「地に足がついた」ものになっていると、多くの場合、より理解しやすくなる。この章では、ブール値と算術式の言語を実装する上で重要な要素を説明する。(後の章で説明する型検査器の実装について考えるつもりのない読者は、この章およびそれ以降の、タイトルに「ML 実装」とある章をすべて読み飛ばしてもよい。)

この章で示すコード(と、本書中の実装に関する章すべてのコード)は、ML 系(Gordon, Milner, and Wadsworth [183])の有名な言語である **OCaml**(Leroy [280]; Cousineau and Mauny [120])で書かれている。**OCaml** 言語全体のごく小さなサブセットのみが使われており、この章の例は他のほとんどの言語に容易に翻訳できるはずである。最も重要な要求は、自動メモリ管理(ガベージコレクション)と、構造データ型に対するパターンマッチを用いて再帰関数を容易に定義できる機能である。他の関数型言語、例えば Standard ML(Milner, Tofte, Harper, and MacQueen [316])、Haskell (Hudak et al. [226]; Thompson [461])や Scheme(Kelsey, Clinger, and Rees [252]; Dybvig [142])(パターンマッチ拡張付き)は良い選択である[†1]。ガベージコレクションはあるがパターンマッチがない Java(Arnold and Gosling [23])や純粋な Scheme のような言語は、これから我々が行うような類いのプログラミングにはいくぶん厳しい。どちらも持たない C(Kernighan and Ritchie [254])のような言語はもっと適さない[†2]。

4.1 構文

最初に、項を表現する OCaml の型を定義する。これは OCaml のデータ型定義機能のおかげで容易である。次の宣言は 17 ページの文法の単純な書き直しである。

```
type term =
    TmTrue of info
  | TmFalse of info
  | TmIf of info * term * term * term
  | TmZero of info
  | TmSucc of info * term
  | TmPred of info * term
  | TmIsZero of info * term
```

TmTrue から TmIsZero までの構築子は、型 term の抽象構文木における異なる種類のノードに名前を付けている。それぞれの of の右に続く型は、その種類のノードに付く部分木の数を指定している。

---

†1 (訳注)より最近の言語としては、F# や Scala も適している。

†2 もちろん言語の好みはさまざまだし、良いプログラマは手にした道具が何であれ、それを使って仕事を完遂できる。どんな言語でも気に入ったものを使うのはあなたの自由だ。しかし注意しよう。(特に)型検査器に必要な記号処理などで、メモリ管理を手動で行うのは、退屈でミスをしやすい作業である。

それぞれの抽象構文木のノードは型 info の値で注釈され、そのノードがどこ（どのソースの何文字目）に由来するのかを表現している。この情報は入力ファイルを読み取る際に構文解析器が作成し、どこでエラーが発生したのかをユーザに知らせるために表示関数が使用する。評価や型検査などの基本的なアルゴリズムを理解するのが目的ならば、この情報もまた省略してしまってかまわない。この情報を含めているのは、単に、自分自身で実装を触ってみたいと望む読者が、本書と全く同じ形のコードを見られるようにするためである。

評価関係の定義では、項が数値かどうかを検査する必要がある。

```
let rec isnumericval t = match t with
    TmZero(_) → true
  | TmSucc(_,t1) → isnumericval t1
  | _ → false
```

これは OCaml のパターンマッチを用いた再帰的定義の典型例である。isnumericval は次のような関数として定義される。TmZero に適用されたときは true を返す。部分木 t1 を持つ TmSucc に適用されたときは t1 が数値かどうか検査するために再帰呼び出しをする。その他の項に適用されたときは false を返す。幾つかのパターンに含まれるアンダースコアは、そこにどんな項があってもマッチする「何でもよい」という印である。これは、最初の二つの節では info 注釈を無視するために、最後の節ではどんな項でもマッチさせるために使われている。rec キーワードによって、これが再帰関数（recursive function）の定義であることをコンパイラに指示している。つまり、関数本体における isnumericval への参照は今定義している最中の関数を指しており、より上にある同じ名前の束縛を指しているわけではない。

上の定義の ML コードは組版のときわずかに「整形」されていることに注意されたい。これは可読性と、ラムダ計算の例との一貫性の、両方を目的としている。例えば、-> の 2 文字の代わりに本物の矢印記号 → を使っている。本書のウェブサイトに、このような整形の完全な一覧がある。

項が値であるかどうかを検査する関数も同様である。

```
let rec isval t = match t with
    TmTrue(_)  → true
  | TmFalse(_) → true
  | t when isnumericval t  → true
  | _ → false
```

三番目の節は「条件付きパターン」である。これは、ブール式 isnumericval t が true になるときに限り、どんな項 t にもマッチする。

4.2 評価

評価関係の実装は、図 3-1 と図 3-2 の単一ステップ評価規則に密接に従っている。これまでに見てきたように、これらの規則は、まだ値でない項に適用されたとき、その項の次の評価ステップを返す**部分**関数を定義する。値に適用されたときは、評価関数は結果を返さない。評価規則を OCaml に翻訳するには、この場合の扱い方を決める必要がある。単純なアプローチの一つは、単一ステップの評価関数 eval1 を、与えられた項に適用できる評価規則がない場合に例外を発生させるように書くことである。（別のやり方として、単一ステップの評価器に term option 型を返させる方法がある。この型は評価が成功したかどうかを示し、成功の場合には結果の項を与える。この方法もうまくいくはずだが、少し余分な手間がかかるだろう。）まず、規則が適用できないときに発生させる例外を定義する。

```
exception NoRuleApplies
```

これで単一ステップの評価器そのものを書ける。

```
let rec eval1 t = match t with
    TmIf(_,TmTrue(_),t2,t3) →
      t2
  | TmIf(_,TmFalse(_),t2,t3) →
      t3
  | TmIf(fi,t1,t2,t3) →
      let t1' = eval1 t1 in
      TmIf(fi, t1', t2, t3)
  | TmSucc(fi,t1) →
      let t1' = eval1 t1 in
      TmSucc(fi, t1')
  | TmPred(_,TmZero(_)) →
      TmZero(dummyinfo)
  | TmPred(_,TmSucc(_,nv1)) when (isnumericval nv1) →
      nv1
  | TmPred(fi,t1) →
      let t1' = eval1 t1 in
      TmPred(fi, t1')
  | TmIsZero(_,TmZero(_)) →
      TmTrue(dummyinfo)
  | TmIsZero(_,TmSucc(_,nv1)) when (isnumericval nv1) →
      TmFalse(dummyinfo)
  | TmIsZero(fi,t1) →
      let t1' = eval1 t1 in
      TmIsZero(fi, t1')
  | _ →
      raise NoRuleApplies
```

幾つかの場所では、既に存在している項を再編成するのではなく、項を一から構築していることに注意されたい。これら新しい項はユーザの元のソースファイルには存在しないので、その info 注釈は無意味である。そのような項の info 注釈には、dummyinfo という定数を用いる。変数名 fi（ファイル情報（file information）の略）はパターン中の info 注釈にマッチさせるために一貫して使われている。

eval1 の定義でもう一つ注意すべき点は、パターンで明示的に when 節を使っていることである。これは図 3-1 と図 3-2 の評価関係のメタ変数名 v や nv による効果を表している。例えば TmPred(_,TmSucc(_,nv1)) の評価に対応する節において、OCaml のパターンの意味のせいで nv1 がどんな項にもマッチしてしまうが、これは我々の望むところではない。when (isnumericval nv1) の追加により規則を制限することで、nv1 にマッチした項が実際に数値であったときのみ起動されるようになっている。（もし望むなら、元の推論規則を ML のパターンと同じスタイルに書き換えることもできた。この場合メタ変数名による暗黙の制約を規則の明示的な付帯条件に変えることになり、

$$\frac{t_1 \text{ は数値}}{\texttt{pred (succ } t_1\texttt{)} \longrightarrow t_1} \qquad \text{(E-PredSucc)}$$

コンパクトさと可読性が少し犠牲になる。）

最後に定義する eval 関数は、項を引数に取り eval1 を繰り返し呼び出すことで正規形を見つけようとする。eval1 が新しい項 t' を返すときは、t' からの評価を続けるために eval を再帰的に呼び出す。最終的に eval1 がどの規則も適用されないポイントに到達したら、例外 NoRuleApplies が発生し、eval はループから抜け出して簡約列の最後の項を返す[†3]。

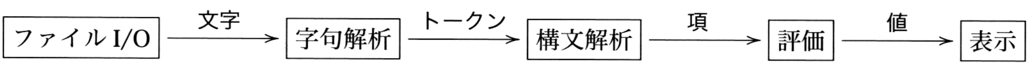
```
let rec eval t =
  try let t' = eval1 t
      in eval t'
  with NoRuleApplies → t
```

当然ながら、この単純な評価器は評価の数学的定義と比較しやすくなるように調整されており、できるだけ速く正規形を見つけるようにはなっていない。むしろ演習 4.2.2 の「大ステップ」評価規則から始めたほうが、いくらか効率の良いアルゴリズムが得られる。

演習 4.2.2. [推奨, ★★★ ↛]　arith の実装における eval 関数の定義を、演習 3.5.17 で導入された大ステップスタイルに変えよ。 □

4.3 最後に

もちろんインタプリタやコンパイラの実装には、たとえ非常に単純な実装であっても、この章で議論した以外の機能も必要である。現実には、評価される前の項はファイル中の文字の列として格納されている。それらは、これまでに見た関数で実際に評価される前に、ファイルシステムから読み込まれ、字句解析器によりトークンのストリームへと処理され、さらに構文解析器によって抽象構文木へと処理される。さらに、評価の後で結果を表示する必要がある。



興味のある読者には、オンラインにあるインタプリタ全体を実装した OCaml コードを見てみることを勧める。

†3 簡単のため eval をこのように書くが、実際のところ ML において try ハンドラを再帰的ループの中に置くのはあまり良いスタイルではない。

演習 4.2.1. [★★]　これはなぜか。eval を書くより良い方法にはどのようなものがあるか。 □

第 5 章

型無しラムダ計算

本章の例は、純粋な型無しラムダ計算 λ（図 5-3)、もしくはブール演算および算術演算で拡張されたラムダ計算λNB（図 3-2）の項である。関連する OCaml 実装は `fulluntyped` である。

この章では、**型無し**ないし**純粋なラムダ計算**の定義と、幾つかの基本的な特徴を概説する。これは今後、本書で説明する型システムのほとんどにおいて基礎となる「計算の土台」である。

1960 年代の中ごろ、Peter Landin は、複雑なプログラミング言語が、その本質的な仕組みを捉えた小さな核となる計算と、その核に翻訳される便利な**派生形式**の集まりとして定式化可能であることに気づいた（Landin [268; 269; 270]。Tennent [457] も参照)。このとき Landin が使った核言語は、1920 年代に Alonzo Church が発明した形式的体系（Church [99; 101]）の一つである、**ラムダ計算**だった。ラムダ計算では、すべての計算が関数定義と関数適用の基本的な演算に帰着される。Landin の洞察や、Lisp に関する John McCarthy の先駆的な業績（McCarthy, Russell, Edwards, et al. [304]; McCarthy [303]）を受けて、ラムダ計算はプログラミング言語の機能の仕様記述や、言語設計と実装、型システムの研究において、広く使われるようになった。この計算体系が重要なのは、**その内部で**計算が記述できる単純なプログラミング言語であると同時に、**それ自身**について厳格な証明が可能な数学的対象ともみなせるからである。

ラムダ計算は同じような目的に用いられている、多数の核となる計算の一つにすぎない。Milner、Parrow、Walker による **π 計算** [312; 310] は、メッセージベースの並行処理言語の意味論を定義するための核言語としてよく使われている。一方、Abadi と Cardelli による**オブジェクト計算** [4] はオブジェクト指向言語の核となる機能を抽出している。本章で解説するラムダ計算のための概念や手法のほとんどは、今挙げたような他の計算にもほぼそのまま適用できるだろう。この線での事例を一つ、第 19 章で説明する。

ラムダ計算はいろいろな方法で拡張できる。まず、核言語で既に模倣可能であっても、数、組、レコードなどの機能のための具体的な特殊構文を加えることはしばしば便利である。さらに興味深いこととして、書き換え可能な参照セルや非局所的な例外の処理といったもっと複雑な機能を加えることもできる。これらの機能は核言語においては、かなり重い変換を用いなければモデル化できない。そのような拡張は ML（Gordon, Milner, and Wadsworth [183]; Milner, Tofte, and Harper [315]; Weis, Aponte, Laville, Mauny, and Suárez [491]; Milner, Tofte, Harper, and MacQueen [316]）や、Haskell（Hudak et al. [226]）、Scheme（Sussman and Steele [450]; Kelsey, Clinger, and Rees [252]）といった言語へとつながっていく。のちの章で見るように、核言語への拡張は、しばしば、型システムへの拡張にも関連する。

5.1 基礎

手続き（もしくは関数）による抽象化は、基本的にすべてのプログラミング言語にとって重要な機能である。同じ計算を何度も何度も書く代わりに、一つ以上の名前の付いた引数を取る、一般的な計算を行う手続きや関数を書く。そして必要に応じて、引数にそれぞれの場合の値を渡し、その関数を具体化する。例えばプログラマには、

```
(5*4*3*2*1) + (7*6*5*4*3*2*1) - (3*2*1)
```

のような長くて繰り返しの多い表現を factorial(5) + factorial(7) - factorial(3) のように書き換える習慣がある。このとき、関数 factorial は以下のように定義される。

```
factorial(n)  =  if n=0 then 1 else n * factorial(n-1)
```

非負の数 n に対し、引数 n で関数 factorial を具体化すると、結果として n の階乗が得られる。「各 n について ... を返す関数」という表現の短縮形として「λn. ...」と書くならば、factorial を

```
factorial  =  λn. if n=0 then 1 else n * factorial(n-1)
```

と定義し直せる。すると factorial(0) は、「関数 (λn. if n=0 then 1 else ...) を引数 0 に適用したもの」を意味する。つまり、「関数の本体 if n=0 then 1 else ... において引数の変数 n を 0 で置き換えたときに結果として得られる値」を指す。それはつまり「if 0=0 then 1 else ...」つまり 1 である。

ラムダ計算（または λ 計算）は、この種の関数定義と関数適用を、最も純粋な形で表現する。ラムダ計算においては、**すべて**が関数である。つまり、関数によって受け付けられる引数はそれ自身関数であり、関数が返す結果もまた関数である。

ラムダ計算の構文はたった三種類の項からなる[†1]。変数 x それ自身は項、項 t_1 から変数 x を抽象化した $\lambda x.t_1$ も項、項 t_1 を他の項 t_2 に適用した $t_1\ t_2$ も項である。これらの項の構成法は以下の文法に要約される。

t ::=	項：
x	変数
λx.t	ラムダ抽象
t t	関数適用

続く小節では、この定義の優れた点を説明する。

抽象構文と具象構文

プログラミング言語の構文を議論するとき、構造の二つのレベル[†2]を区別することが役に立つ。言語の**具象構文**（または**表層構文**）とはプログラマが直接読み書きする文字列のことを指す。**抽象構文**は、プログラムのより単純な内部的表現であり、ラベル付き木（**抽象構文木**（abstract syntax tree）もしくは **AST** と呼ばれる）として表現される。木による表現は、項の構造を直ちに明らかにし、厳格な言語定義（およびそれに関する証明）とコンパイラ・インタプリタの内部の両方に関連する、複雑な操作に自然に合致する。

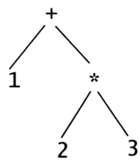
具象構文から抽象構文への変換は、二つの段階からなる。まず、**字句解析器**が、プログラマの書いた文字列を、識別子、予約語、定数、区切り文字といった**トークン**の列に変換する。字句解析器は、コメントを除去し、空白や大文字小文字に関する慣習や、数値や文字列の定数のフォーマットを処理する。次に、**構文解析器**がトークンの列を抽象構文木に変換する。このとき、演算子の**優先順位**や**結合規則**といったさまざまな慣習を用いることで、複合的な式の構造を明示的に表現するための括弧で表層プログラムを汚してしまう必要性が少なくなる。例えば、* は + よりも結合が強い

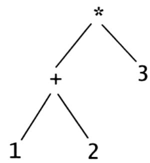
---

†1 **ラムダ項**という用語は、λ で始まる項（しばしば**ラムダ抽象**と呼ばれる）だけでなく、ラムダ計算における任意の項を表す。

†2 本格的な言語の定義はさらに多くのレベルに分かれることもある。例えば Landin に従い、一部の言語機構の振る舞いを派生形式として定義し、より基本的な他の機能の組み合わせに変換することはしばしば有用である。このとき、核となる機能だけを含んだ、制限された部分言語は、**内部言語**と呼ばれる。一方、すべての派生形式を含んだ完全な言語は**外部言語**と呼ばれる。外部言語から内部言語への変換は、構文解析の後、（少なくとも概念的には）独立したパスで行われる。派生形式は 11.3 節で議論する。

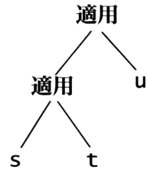
ので、構文解析器は括弧のない式 1+2*3 を抽象構文木として、左図のように解釈する（右図のようには解釈しない）。



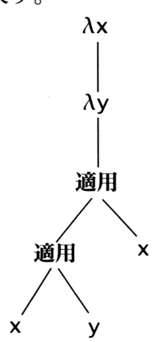


本書で注意を向けるのは、具象構文ではなく抽象構文である。例えば上記のラムダ項の文法は、トークンや文字の列ではなく、正しい木構造を記述するものと理解されるべきである。もちろん、例、定義、定理、および証明の中で項を書くときには、具体的な文字列表記で項を表現する必要があるが、裏では常に、根底にある抽象構文木を念頭に置く。

たくさんの括弧を書かなくてもよいように、一列にラムダ項を書く際に二つの慣習を採用する。まず一つ目に、関数適用は左結合とする。つまり、s t u は (s t) u と同じ木を表す。



二つ目に、抽象の本体はできる限り右側へ拡大する。したがって、例えば、λx. λy. x y x は以下に示す λx. (λy. ((x y) x)) と同じ木を表す。



変数とメタ変数

上の構文定義におけるもう一つの微妙な点は、メタ変数の使用に関するものである。任意の項を表すために、メタ変数 t（および s、u）を（添字付きおよび添字なしで）使い続けることにする[†3]。同様に、x（および y、z）は任意の変数を表す。ここで、x は変数上を動くメタ変数であることに注意されたい。さらに悪いことに、短い名前は数が限られるので、対象言語の変数としても、x や y を使いたい。だが、そのような場合には常に、どちらがどちらであるか、文脈から明らかだろう。例えば、「項 λx. λy. x y は z = x と s = λy. x y の下で λz.s という形である」のような文中では、名前 z や s はメタ変数であり、一方 x や y は対象言語の変数である。

[†3] もちろん、この章では、t は算術式の上ではなく、ラムダ項の上を動く。本書を通じて、t は常にそのときどきに議論している計算体系の項の上を動く。各々の章の最初のページの脚注で、どんな体系かを指定する。

スコープ

ラムダ計算の構文について説明しなければならない最後の点は、変数の**スコープ**である。

抽象 λx.t の本体 t の中に変数 x が現れるとき、その x の出現は**束縛**されているという（より正確には、**この**抽象によって束縛されているという。同じように、λx は t をスコープとする**束縛子**であるという）。x を囲む抽象によって束縛されていない場所に x が現れるとき、その x の出現は**自由**であるという。例えば、x y や λy. x y における x の出現は自由だが、λx.x や λz. λx. λy. x (y z) における x の出現は束縛されている。(λx.x) x においては、一つ目の x の出現は束縛されているが、二つ目の出現は自由である。

自由変数のない項は**閉じている**という。閉じた項は**コンビネータ**とも呼ばれる。最も単純なコンビネータは、**恒等関数**（identity function）と呼ばれる次の関数である。

```
id = λx.x;
```

これは、何もせず引数を返す関数である。

操作的意味論

純粋なラムダ計算は、組み込みの定数やプリミティブな演算子を持たない。例えば、数値や、算術演算、条件式、レコード、ループ、列、入出力などはない。「計算」が意味するのはただ一つ、引数に対する関数の適用である（引数自身も関数である）。計算の各ステップでは、左側が抽象であるような関数適用を、その抽象の本体に含まれる束縛変数に関数適用の右側の項を代入したもので書き換える。図式的には、

$$(\lambda x.\ t_{12})\ t_2 \longrightarrow [x \mapsto t_2]t_{12}$$

と書く。ここで $[x \mapsto t_2]t_{12}$ は、「t_{12} の中の x の自由な出現をすべて t_2 で置き換えることによって得られる項」を意味する。例えば、項 (λx.x) y は y と評価される。項 (λx. x (λx.x)) (u r) は u r (λx.x) と評価される。Church に従い、$(\lambda x.\ t_{12})\ t_2$ という形の項を**簡約基**（redex、“reducible expression”）と呼び、上記の規則で簡約基を書き換える操作のことを**ベータ簡約**と呼ぶ。

ラムダ計算のための評価戦略には数種類あり、長年にわたってプログラミング言語の設計者や理論家が研究してきた。それぞれの戦略は、次の評価ステップで項の中のどの簡約基が評価されうるかを定義する[†4]。

- **完全ベータ簡約**の下では、任意の簡約基がいつでも簡約されうる。各ステップで評価している項の中の任意の箇所から簡約基を取り出して、簡約する。例えば、項

  (λx.x) ((λx.x) (λz. (λx.x) z))

  を考える。この項は、id (id (λz. id z)) のようにもっと読みやすく書くこともできる。この項は、三つの簡約基を含む。

  id (id (λz. id z))
  id (id (λz. id z))
  id (id (λz. id z))

  完全ベータ簡約においては、例えば最も内部の簡約基から始めて、次に真ん中、そして最も外側というように選ぶこともできる。

---

†4 「簡約」と「評価」を同義に用いる人もいれば、「値」と「簡約」の概念に関する戦略だけに「評価」を用いる人もいる。

```
     id (id (λz. id z))
→   id (id (λz.z))
→   id (λz.z)
→   λz.z
↛
```

- **正規順序**戦略の下では常に、最も左で最も外側の簡約基が最初に簡約される。この戦略の下では、上述の項は次のように簡約される。

```
     id (id (λz. id z))
→   id (λz. id z)
→   λz. id z
→   λz.z
↛
```

  この戦略（および次に述べる戦略）の下では、評価関係は実際には部分関数である。つまり、各項 `t` は 1 ステップで高々一つの項 `t′` に評価される。
- **名前呼び**戦略はさらに制限が強く、抽象の内部での簡約を許さない。同じ項から始めると、最初の二つの簡約は正規順序と同じように行うが、三つ目の簡約の前で停止し、`λz. id z` を正規形とみなす。

```
     id (id (λz. id z))
→   id (λz. id z)
→   λz. id z
↛
```

  名前呼びの変種は、幾つかのよく知られたプログラミング言語、特に Algol-60（Naur et al. [334]）と Haskell（Hudak et al. [226]）で使用されている。Haskell は実際には**必要呼び**（Wadsworth [485]; Ariola et al. [22]）として知られる、最適化された変種を用いている。必要呼びにおいては、引数が用いられるたびに再評価するのではなく、最初に評価されたときにその引数の出現をすべて評価結果の値に置き換えるので、二回目以降に再評価しなくて済む。この戦略では項の実行時の表現において、ある種の共有を保持する必要がある。事実上、抽象構文木ではなく抽象構文**グラフ**上での簡約関係である。
- ほとんどの言語は**値呼び**戦略を用いる。この評価戦略では、最も外側の簡約基を簡約するの**だが**、右側が既に**値**（計算が終わっていてそれ以上簡約できない閉じた項）に簡約されている簡約基だけを簡約する[†5]。この戦略下では、先の例を以下のように簡約する。

```
     id (id (λz. id z))
→   id (λz. id z)
→   λz. id z
↛
```

  値呼び戦略は**正格**である。これは、関数の引数が関数の本体で使われるかどうかに関係なく常に評価されるという意味である。一方、名前呼びや必要呼びといった**非正格な**（もしくは**遅延評価**）戦略は、引数が実際に使われるときにだけ評価される。

[†5] 現在の必要最低限の計算においては、値はラムダ抽象だけである。より強力な計算では、数やブール値の定数、文字列、値の組、レコード、配列などの他の値も扱うことになる。

実際には、評価戦略の選択が型システムを論じる上で違いを生むことはほとんどない。さまざまな型付けの機能が生まれる元となった課題や、それらを解決するのに使われる技術は、どの戦略でもほとんど同じである。本書では値呼びを用いる。理由は二つある。よく知られた言語のほとんどで採用されている戦略であり、また、例外（第 14 章）や参照（第 13 章）といった機能によって拡張するのが最も簡単な戦略であるためである。

5.2 ラムダ計算でのプログラミング

ラムダ計算は、定義の小ささとは裏腹に、非常に強力である。本節ではラムダ計算におけるプログラミングの標準的な例を幾つか説明する。その例によって、ラムダ計算が単体で本格的なプログラミング言語であるといいたいわけではない。広く普及しているどの高水準言語でも、同じことはより明快かつ効率的に達成できる。むしろ、計算体系の感覚をつかむための準備体操のつもりである。

複数の引数

まずはじめに、ラムダ計算は複数引数の関数を組み込みでサポートしていないことに注目してほしい。もちろん、サポートすることは難しくないだろうが、関数を返り値として返す**高階関数**を用いて同等の効果を達成するほうが容易である。`s` が二つの自由変数 `x` と `y` を含む項であるとし、引数の各組 `(v,w)` に対して `s` の中の `x` に `v` を、`y` に `w` を代入した結果を返す関数 `f` を書きたいとしよう。`f = λ(x,y).s` と書くのは、より強力なプログラミング言語なら可能だが、ここでは代わりに `f = λx.λy.s` と書く。つまり `f` は、`x` に対して値 `v` が与えられたとき、`y` に対して値 `w` が与えられたときに望む結果を返す関数、を返す関数である。次に、複数の引数に対して一度に一つずつ `f` を適用する。`f v w`（つまり、`(f v) w`）は、`((λy.[x ↦ v]s) w)` に簡約され、それから `[y ↦ w][x ↦ v]s` に簡約される。複数の引数を取る関数を高階関数に変換することを**カリー化**という。これは Church と同時代の Haskell Curry にちなむ。

Church ブール値

ラムダ計算で容易に表現できるもう一つの言語機能は、ブール値と条件式である。`tru` と `fls` という項を以下のように定義する。

```
tru = λt. λf. t;
fls = λt. λf. f;
```

（短縮形の綴りにしたのは、第 3 章からの、プリミティブなブール定数 `true` および `false` との混乱を避けることを意図している。）

項 `tru` および `fls` は、これらの項を用いてブール値の真偽を判定する操作を行えるという意味で、「真」と「偽」を**表現している**とみなせる。特に、`test b v w` は `b` が `tru` のとき `v` へ簡約され、`b` が `fls` のとき `w` に簡約されるという性質を持つコンビネータ `test` を、関数適用を用いて定義できる。

```
test = λl. λm. λn. l m n;
```

実のところ、`test` コンビネータは大したことをしていない。`test b v w` は単に `b v w` に簡約されるだけである。事実上、ブール値 `b` 自身が条件式である。すなわち二つの引数を取り、`tru` ならば一つ目を、`fls` ならば二つ目を返す。例えば、`test tru v w` は以下のように簡約される。

```
      test tru v w
  =   (λl. λm. λn. l m n) tru v w   定義より
  ⟶   (λm. λn. tru m n) v w         下線部の簡約基を簡約した
  ⟶   (λn. tru v n) w               下線部の簡約基を簡約した
  ⟶   tru v w                       下線部の簡約基を簡約した
  =   (λt.λf.t) v w                 定義より
  ⟶   (λf. v) w                     下線部の簡約基を簡約した
  ⟶   v                             下線部の簡約基を簡約した
```

論理積などのブール演算子も関数として定義できる。

```
and = λb. λc. b c fls;
```

つまり and は、二つのブール値 b と c を受け取り、b が tru のときには c を返し、b が fls のときには fls を返す関数である。したがって and b c は、b と c が共に tru のときは tru となり、b か c のいずれかが fls であれば fls となる。

```
  and tru tru;
▸ (λt. λf. t)

  and tru fls;
▸ (λt. λf. f)
```

演習 5.2.1. [★]　論理演算の or 関数と not 関数を定義せよ。 □

二つ組

ブール値を用いると、二つの値の組を項として表現できる。

```
pair = λf.λs.λb. b f s;
fst = λp. p tru;
snd = λp. p fls;
```

つまり pair v w は、ブール値 b に対して適用されたとき、b を v と w に適用する関数である。ブール値の定義から、この関数適用は b が tru のときに v を、b が fls のときに w を返す。よって、一番目と二番目の射影関数 fst と snd は、適切なブール値を与えるだけで実装できる。fst (pair v w) $\longrightarrow^*$ v は以下のように計算される。

```
      fst (pair v w)
  =   fst ((λf. λs. λb. b f s) v w)   定義より
  ⟶   fst ((λs. λb. b v s) w)         下線部の簡約基を簡約した
  ⟶   fst (λb. b v w)                 下線部の簡約基を簡約した
  =   (λp. p tru) (λb. b v w)         定義より
  ⟶   (λb. b v w) tru                 下線部の簡約基を簡約した
  ⟶   tru v w                         下線部の簡約基を簡約した
  ⟶*  v                               前回と同様
```

Church 数

ラムダ項で数を表現することは、これまで見てきたものより少しだけややこしい。**Church 数** c_0、c_1、c_2、…を以下のように定義する。

```
c0 = λs. λz. z;
c1 = λs. λz. s z;
c2 = λs. λz. s (s z);
c3 = λs. λz. s (s (s z));
```

以下同様。つまり数 n は、二つの引数 s と z（後者関数（successor）とゼロの意）を取り、s を z に n 回適用するコンビネータ c_n によって表現される。ブール値や二つ組と同様、この表現では数は動作を伴うものになる。数 n は何かを n 回する関数として表現される。いわば動く一進数である。

（c_0 と fls が実際には同じ項であることに気づいている読者もいるかもしれない。似たような「語呂合わせ」はアセンブリ言語でもよく用いられる。同じビットパターンでも、例えば整数、浮動小数、アドレス、4 文字の文字列など、解釈の仕方に依存してさまざまな値を表現しうる。C などの低水準言語でも 0 と false を同等とみなすことがある。）

Church 数における後者関数は次のように定義できる。

```
scc = λn. λs. λz. s (n s z);
```

項 scc は、Church 数 n を取り、別の Church 数を返すコンビネータである。つまり、引数に s と z を取り、s を繰り返し z に適用する関数を返す。まず s と z を n への引数として渡し、さらにもう一回 s を結果へ明示的に適用することによって、s を z へ正しい回数だけ適用したものを得られる。

演習 5.2.2. [★★]　Church 数における後者関数を定義する別の方法を見つけよ。　□

同様に Church 数の加算は、二つの Church 数 m と n を引数に取り、別の Church 数を返す項 plus によって計算できる。返り値の Church 数（つまり、関数）は、s と z を引数に取り、（n への引数として s と z を渡すことにより）s を z に n 回適用し、さらにその結果へ s を m 回適用することで新たな Church 数を返す。

```
plus = λm. λn. λs. λz. m s (n s z);
```

乗算の実装には別の仕掛けを用いる。plus は引数を一度に一つ受け取るので、一つの引数 n だけに適用すると、与えられた任意の引数に n を足す関数が得られる。この関数に第一引数として m を渡し、第二引数として c_0 を渡すことで、「引数に n を足す関数をゼロに m 回適用すること」、すなわち、「n を m 回コピーして合計すること」という意味になる。

```
times = λm. λn. m (plus n) c0;
```

演習 5.2.3. [★★]　plus を用いずに Church 数の乗算を定義できるか。　□

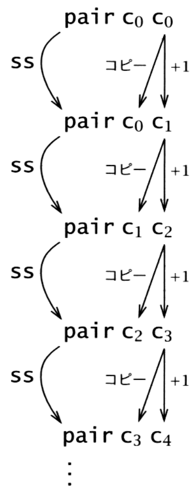
演習 5.2.4. [推奨, ★★]　ある数を他の数で冪乗する項を定義せよ。　□

Church 数がゼロかどうかを判定するには、その情報を返すような適切な引数を二つ見つけなければならない。具体的には、zz に ss を一回以上適用すると fls、全く適用しないと tru を返すような二つの項 zz と ss を見つけ、それらに対して判定したい Church 数を適用する。明らかに、zz は単なる tru とすべきである。ss には、引数を無視して常に fls を返す関数を用いる。

```
iszro = λm. m (λx. fls) tru;

iszro c1;
▸ (λt. λf. f)

iszro (times c0 c2);
▸ (λt. λf. t)
```



▲ 図 5-1. 前者関数の「内部ループ」

驚くことに、Church 数を使った減算は、加算よりもかなり難しい。以下のかなりややこしい「前者関数」を用いれば可能ではある。この関数は c_0 を引数として与えると c_0 を返し、c_{i+1} を与えると c_i を返す。

```
zz = pair c₀ c₀;
ss = λp. pair (snd p) (plus c₁ (snd p));
prd = λm. fst (m ss zz);
```

この定義は、m を関数として用いて、関数 ss の m 個のコピーを初期値に適用することで動作する。ss の各コピーはその引数として数の二つ組 pair c_i c_j を受け取り、結果として pair c_j c_{j+1} を返す (図 5-1 を見よ)。よって ss を m 回 pair c_0 c_0 に適用することで、$m = 0$ のときには pair c_0 c_0 を、m が正のときには pair c_{m-1} c_m を得る。どちらの場合でも、第一要素が m の前者値となっている。

演習 5.2.5. [★★] prd を使って減算の関数を定義せよ。 □

演習 5.2.6. [★★] prd c_n を計算するために、(n の関数として) おおよそ何ステップの評価が必要か。 □

演習 5.2.7. [★★] 二つの数の等価性を調べて、Church ブール値を返す関数 equal を書いてみよ。この equal は例えば、以下のように動作する。

```
  equal c₃ c₃;
▶ (λt. λf. t)

  equal c₃ c₂;
▶ (λt. λf. f)
```

□

リスト、木、配列、バリアント、レコードといった他の一般的なデータ型は同様の方法で表現できる。

演習 5.2.8. [推奨, ★★★] ラムダ計算において、リストはそれ自身の fold 関数として表現できる (この関数は OCaml では fold_right という名前である。reduce と呼ばれることもある)。例え

ばリスト [x,y,z] は、二つの引数 c と n を受け取り、c x (c y (c z n))) を返す関数になる。nil の表現はどうなるだろうか。要素 h とリスト（すなわち fold 関数）t を受け取り、h を t の前に付加したリストに対応する表現を返す関数 cons を書け。リストを引数に取る isnil 関数と head 関数を書いてみよ。最後にこのリスト表現に対して、tail 関数を書いてみよ（これはかなり難しく、数における prd の定義と似たような仕掛けを必要とする）。 □

計算体系の拡張

ブール値や数とその演算が、純粋なラムダ計算の中で表現できることを見てきた。実際、厳密に言えば、およそ必要なプログラミングはすべて、この純粋な体系の中で可能である。しかし例を扱う際、プリミティブなブール値や数（場合によっては他のデータ型）も含まれているとしばしば便利である。正確にどちらの体系で議論しているかをはっきりさせる必要がある場合、図 5-3 で定義される純粋なラムダ計算に対しては記号 λ を用い、図 3-1 と図 3-2 で定義したブール値や算術式を伴う拡張された体系に対しては記号λNB を用いる。

λNB では事実上、二つの異なるブール値の実装と二つの異なる数の実装があり、プログラムを書く際に選択できる。すなわち、本物の値と、この節で解説した表現の二つである。もちろん、この二つを相互に変換することは簡単である。Church ブール値をプリミティブなブール値に変えるには、Church ブール値を true と false に適用すればよい。

```
realbool = λb. b true false;
```

逆向きには if 式を用いる。

```
churchbool = λb. if b then tru else fls;
```

高水準な演算に対してこれらの変換を行うこともできる。本物のブール値を返す Church 数の等価関数は以下のようになる。

```
realeq = λm. λn. (equal m n) true false;
```

同様に、Church 数に succ と 0 を適用することで、対応するプリミティブな数に変換できる。

```
realnat = λm. m (λx. succ x) 0;
```

succ は単体では構文的な意味をなさないので、m を succ に直接適用することはできない。算術式の構文を定義した方法により、succ は何かに適用されなければならない。そこで、引数の succ を返すだけの小さな関数の中に succ を格納することで、この問題を避けている。

事例においてプリミティブなブール値や数が役立つ理由は、主として評価順序に関係がある。例えば、項 scc c_1 を考えてみよ。上の議論から、この項が Church 数 c_2 に評価されるはずだと期待するかもしれない。実際には、そうはならない。

```
scc c₁;
▸ (λs. λz. s ((λs'. λz'. s' z') s z))
```

この項は簡約基を持ち、これを簡約すれば（2 ステップで）c_2 となるが、この簡約基はラムダ抽象の中にあるので、値呼び評価の規則ではまだ簡約できない。

これは本質的な問題ではない。scc c_1 の評価から得られる項は、明らかに c_2 と**振る舞い等価である**。すなわち、任意の引数の二つ組 v と w に対してこの項を適用すると、v と w に c_2 を適用した場合と同じ結果となる。それでもやはり、残っている計算のせいで、scc 関数が期待通りに振る舞っていることを確かめづらい。より複雑な算術演算では、さらに確かめづらくなる。例えば times c_2 c_2 は、c_4 ではなく、以下の巨大な怪物になる。

```
   times c₂ c₂;
 ▶ (λs.
      λz.
        (λs'. λz'. s' (s' z')) s
        ((λs'.
            λz'.
              (λs". λz". s" (s" z")) s'
              ((λs". λz".z") s' z'))
         s
         z))
```

この項が c_4 のように振る舞うことを確認する一つの方法は、等価性をテストしてみることである。

```
   equal c₄ (times c₂ c₂);
 ▶ (λt. λf. t)
```

しかし、times c_2 c_2 をプリミティブな数に変換するほうが直接的である。

```
   realnat (times c₂ c₂);
 ▶ 4
```

この変換は、times c_2 c_2 がさらに必要とする二つの引数を補い、中に含まれる潜在的な計算をすべて実行させる作用がある。

再帰

評価関係の下でステップを進めることができない項のことを、**正規形**と呼んだことを思い出そう。興味深いことに、正規形へ評価することができない項が存在する。例えば、**発散**コンビネータ

```
   omega = (λx. x x) (λx. x x);
```

は、簡約基を一つだけ含み、その簡約基を簡約すると、また omega そのものになるのである。項が正規形を持たないとき、その項は**発散**するという。

omega コンビネータには、**不動点コンビネータ**と呼ばれる有用な一般化がある[†6]。不動点コンビネータを用いると factorial のような再帰関数を定義できる[†7]。

```
   fix = λf. (λx. f (λy. x x y)) (λx. f (λy. x x y));
```

omega と同じように、fix コンビネータは入り組んだ反復的構造を持つ。定義を読んだだけで理解することは難しい。その振る舞いを直観的に理解する最善の方法はおそらく、具体的な例に対してどのように作用するかを観察することである[†8]。h = ⟨h を含む本体⟩ の形をした再帰関数定義を書きたいとしよう。つまり、40 ページの factorial の定義のように、= の右辺の項で、現在定義している関数そのものを用いるような定義を書きたい。ここで意図しているのは、再帰的な定義は出現する箇所で定義が「展開」されるべきということである。例えば factorial の定義は、直観的に

```
   if n=0 then 1
   else n * (if n-1=0 then 1
             else (n-1) * (if (n-2)=0 then 1
                           else (n-2) * ...))
```

†6 しばしば**値呼び Y コンビネータ**と呼ばれる。Plotkin [381] は Z と呼んでいる。

†7 注意として、より単純な名前呼びでの不動点コンビネータ

```
   Y = λf. (λx. f (x x)) (λx. f (x x))
```

は、値呼びでは使えない。任意の g に対して Y g が発散するからである。

†8 fix の定義を第一原理（Friedman and Felleisen [162] の Chapter 9 など）から導出することもできるが、その導出もかなり入り組んでいる。

であるか、または Church 数を用いて

```
if realeq n c₀ then c₁
else times n (if realeq (prd n) c₀ then c₁
              else times (prd n)
                        (if realeq (prd (prd n)) c₀ then c₁
                         else times (prd (prd n)) ...))
```

であるかだろう。fix コンビネータを用いればこの挙動が得られる。まず g = λf.⟨f を含む本体⟩ と定義し、次に h = fix g と定義する。例えば階乗関数は

```
g = λfct. λn. if realeq n c₀ then c₁ else (times n (fct (prd n)));
factorial = fix g;
```

として定義できる。図 5-2 は、評価の最中に、項 factorial c_3 がどうなるかを示している。計算がうまくいく鍵は、fct n $\longrightarrow^*$ g fct n である。つまり fct は一種の「自己複製子」で、引数に適用された際、それ自身と n とを引数として g に与える。g の本体の中で g の第一引数が現れたときは常に、fct の別のコピーが得られる。それを引数に適用すると、再度自分自身とその引数を g に渡す。以下同様である。fct を用いた再帰呼び出しでは常に、g の本体のもう一つのコピーを展開し、それに対し、再度展開する準備のできた新しい fct のコピーを据える。

演習 5.2.9. [★] なぜ g の定義の中で、Church ブール値に関する Church ブール関数 test ではなく、プリミティブな if を使ったか。if ではなく test を使った factorial を定義する方法を示せ。 □

演習 5.2.10. [★★] プリミティブな自然数を、対応する Church 数に変換する関数 churchnat を定義せよ。 □

演習 5.2.11. [推奨, ★★] fix と、演習 5.2.8 でのリストの表現とを用いて、Church 数のリストの総和を計算する関数を書け。 □

表現

例の説明を終えてラムダ計算の形式的な定義に進む前に、最後にもう一つの質問について考えてみたい。Church 数が普通の数を**表現**するとは、厳密には何を意味するのか？

これに答えるにはまず、普通の数とは何かということを思い出す必要がある。普通の数を定義するための（等価な）方法はたくさんある。本書が（図 3-2 で）選んでいる方法は、以下を定義する方法である。

- 定数 0
- 数をブール値に写す演算 iszero
- 数を数に写す二つの演算 succ と pred

算術演算の振る舞いは、図 3-2 に示す評価規則によって定義される。これらの規則から、例えば、3 は 2 の後者値であり、iszero 0 は真であるとわかる。

数の Church 表現は、上のそれぞれを、ラムダ項（つまり関数）として表現している。

- 項 c_0 は数 0 を表現している。
  48 ページで見たように、項による数の「非標準的表現」もある。例えば λs. λz. (λx. x) z は、c_0 と振る舞い等価であり、これも 0 を表現している。

factorial c_3

$=$ fix g c_3

$\longrightarrow$ h h c_3

ただし h = λx. g (λy. x x y)

$\longrightarrow$ g fct c_3

ただし fct = λy. h h y

$\longrightarrow$ (λn. if realeq n c_0
then c_1
else times n (fct (prd n)))
c_3

$\longrightarrow$ if realeq c_3 c_0
then c_1
else times c_3 (fct (prd c_3))

$\longrightarrow^*$ times c_3 (fct (prd c_3))

$\longrightarrow^*$ times_c_3 (fct (prd c_3))

ただし times_c_3 は times c_3 と振る舞い等価である

$\longrightarrow^*$ times_c_3 (fct c'_2)

ただし c'_2 は c_2 と振る舞い等価である

$\longrightarrow^*$ times_c_3 (g fct c'_2)

$\longrightarrow^*$ times_c_3 (times_c'_2 (g fct c'_1))

ただし times_c'_2 は times c'_2 と振る舞い等価であり、c'_1 は c_1 と振る舞い等価である
(g fct c'_2 に対して同じ計算を繰り返すことによって)

$\longrightarrow^*$ times_c_3 (times_c'_2 (times_c'_1 (g fct c'_0)))

ただし times_c'_1 は times c'_1 と振る舞い等価であり、c'_0 は c_0 と振る舞い等価である
(同様)

$\longrightarrow^*$ times_c_3 (times_c'_2 (times_c'_1 (if realeq c'_0 c_0 then c_1
else ...)))

$\longrightarrow^*$ times_c_3 (times_c'_2 (times_c'_1 c_1))

$\longrightarrow^*$ c'_6

ただし c'_6 は c_6 と振る舞い等価である

▲ 図 5-2. factorial c_3 の評価

- 項 scc と prd は算術演算 succ と pred を表現している。すなわち、t が数 n の表現ならば、scc t は n + 1 の表現に評価され、prd t は n − 1 の表現に評価（n が 0 のときは 0 の表現に評価）されるという意味である。
- 項 iszro は演算 iszero を表現している。すなわち、t が 0 の表現ならば、iszro t は true に評価される[†9]。t が 0 以外の数を表現するならば、iszro t は false と評価される。

以上をまとめて、数に関する複雑な計算をしてブール値を返すプログラムを丸ごと一つ手に入れたとしよう。すべての数と算術演算を、それらを表現するラムダ項に置き換えてからプログラムを評価したとしても、同じ結果を得る。したがって、プログラムの全体的な結果に関する効果という意味では、本当の数と Church 数としての表現との間に観測可能な違いはない。

[†9] 厳密に言うと、定義した通りに、iszro t は項としての true **の表現**に評価されるのだが、今の議論を簡単にするために違いを無視することにしよう。数の場合と同様の議論により、Church ブール値はどういう意味で本当のブール値を表現しているかを説明できる。

5.3 形式的議論

これ以降本章では、ラムダ計算の構文と操作的意味論をより詳細に見ていく。必要な構造のほとんどは、第 3 章で扱ったものと非常によく似ている（同じ構造をそのまま繰り返すのを避けるために、ここではブール値や数を持たない純粋なラムダ計算を対象とする）。しかし、項を変数へと代入する操作には驚くほど巧妙な部分がある。

構文

第 3 章と同じく、項を定義する抽象構文（40 ページ）は、帰納的に定義された抽象構文木の集合に対する簡易な表記として扱う。

定義 5.3.1.［項］　$\mathcal{V}$ を変数名の可算集合とする。項の集合は以下を満たす最小の集合 $\mathcal{T}$ である。

(1) 任意の $\mathtt{x} \in \mathcal{V}$ について $\mathtt{x} \in \mathcal{T}$
(2) $\mathtt{t}_1 \in \mathcal{T}$ かつ $\mathtt{x} \in \mathcal{V}$ ならば、$\lambda\mathtt{x}.\mathtt{t}_1 \in \mathcal{T}$
(3) $\mathtt{t}_1 \in \mathcal{T}$ かつ $\mathtt{t}_2 \in \mathcal{T}$ ならば、$\mathtt{t}_1\ \mathtt{t}_2 \in \mathcal{T}$　□

項 t の**サイズ**は、定義 3.3.2 で算術式にしたのと全く同じように定義できる。より面白いのは、ラムダ項に自由に現れる変数集合の単純な帰納的定義ができることである。

定義 5.3.2.　項 t の**自由変数**（free variable）の集合は、$FV(\mathtt{t})$ と書かれ、以下のように定義できる。

$$
\begin{aligned}
FV(\mathtt{x}) &= \{\mathtt{x}\} \\
FV(\lambda\mathtt{x}.\mathtt{t}_1) &= FV(\mathtt{t}_1) \setminus \{\mathtt{x}\} \\
FV(\mathtt{t}_1\ \mathtt{t}_2) &= FV(\mathtt{t}_1) \cup FV(\mathtt{t}_2)
\end{aligned}
$$
□

演習 5.3.3.［★★］　任意の項 t について $|FV(\mathtt{t})| \leq size(\mathtt{t})$ が成り立つことを注意深く証明せよ。□

代入

代入の操作は詳細に見ると実はかなり技巧的である。本書では現に、異なった目的に最適化された二通りの定義を用いる。最初のものは、本節で導入するもので、コンパクトかつ直観的である。また、例に対してよく機能し、数学的な定義や証明にも適している。もう一つのものは、第 6 章で考えるもので、項の別の表現である「de Bruijn 表現」に基づく、より重い表記を用いている。この表記では、名前付きの変数を、数によるインデックスに置き換える。重い表記ではあるものの、後の章で扱う ML による実際の実装には便利である。

代入の正しい定義を示す前に、説明のため、誤った試みを幾つか挙げる。最初は、最も素朴な再帰的定義を試してみよう。（形式的に言えば、関数 $[\mathtt{x} \mapsto \mathtt{s}]$ を引数 t に関する帰納法によって定義する。）

$$
\begin{aligned}
[\mathtt{x} \mapsto \mathtt{s}]\mathtt{x} &= \mathtt{s} \\
[\mathtt{x} \mapsto \mathtt{s}]\mathtt{y} &= \mathtt{y} && \mathtt{x} \neq \mathtt{y}\ \text{の場合} \\
[\mathtt{x} \mapsto \mathtt{s}](\lambda\mathtt{y}.\mathtt{t}_1) &= \lambda\mathtt{y}.\ [\mathtt{x} \mapsto \mathtt{s}]\mathtt{t}_1 \\
[\mathtt{x} \mapsto \mathtt{s}](\mathtt{t}_1\ \mathtt{t}_2) &= ([\mathtt{x} \mapsto \mathtt{s}]\mathtt{t}_1)\ ([\mathtt{x} \mapsto \mathtt{s}]\mathtt{t}_2)
\end{aligned}
$$

この定義はほとんどの例に対してうまく機能する。例えば、

$$[\mathtt{x} \mapsto (\lambda\mathtt{z}.\ \mathtt{z}\ \mathtt{w})](\lambda\mathtt{y}.\mathtt{x}) = \lambda\mathtt{y}.\lambda\mathtt{z}.\ \mathtt{z}\ \mathtt{w}$$

のようになり、これは代入に直観的に期待する振る舞いに合っている。しかし、運の悪い束縛変数名の選択をすると、この定義は破綻してしまう。例えば以下の例を考える。

$$[\mathtt{x} \mapsto \mathtt{y}](\lambda\mathtt{x.x}) = \lambda\mathtt{x.y}$$

これは関数抽象に対する「**束縛変数名は問わない**」という基本的な直観、すなわち「恒等関数は λx.x と書こうと λy.y と書こうと、あるいは λfranz.franz と書こうと、完全に同じものだ」という直観に矛盾している。もし、代入に対してこれらが完全に同じように振る舞わないとしたら、簡約に対しても同じように振る舞わない。これは誤っているように思われる。

代入のこの素朴な定義で犯した最初の誤りは、明らかに、変数 x の項 t 内での**自由な**出現と、**束縛された**出現を区別しなかったことである。前者の出現は代入の過程で置換されるべきだが、後者の出現は置換されるべきではない。項 t 中で名前 x を束縛するラムダ抽象に遭遇したら、代入操作はそこで止まるべきである。そのようにしたのが以下に試みる定義である。

$$\begin{array}{lcll}
[\mathtt{x} \mapsto \mathtt{s}]\mathtt{x} & = & \mathtt{s} & \\
[\mathtt{x} \mapsto \mathtt{s}]\mathtt{y} & = & \mathtt{y} & \mathtt{y} \neq \mathtt{x}\text{ の場合} \\
[\mathtt{x} \mapsto \mathtt{s}](\lambda\mathtt{y.t_1}) & = & \begin{cases} \lambda\mathtt{y.\ t_1} & \mathtt{y} = \mathtt{x}\text{ の場合} \\ \lambda\mathtt{y.\ }[\mathtt{x} \mapsto \mathtt{s}]\mathtt{t_1} & \mathtt{y} \neq \mathtt{x}\text{ の場合} \end{cases} \\
[\mathtt{x} \mapsto \mathtt{s}](\mathtt{t_1\ t_2}) & = & ([\mathtt{x} \mapsto \mathtt{s}]\mathtt{t_1})\ ([\mathtt{x} \mapsto \mathtt{s}]\mathtt{t_2}) &
\end{array}$$

これはさっきより良いが、まだ完全に正しくはない。例えば、項 λz.x 中の変数 x に対して項 z を代入しようとしたらどうなるだろうか。

$$[\mathtt{x} \mapsto \mathtt{z}](\lambda\mathtt{z.x}) = \lambda\mathtt{z.z}$$

今度は、さっきとは正反対の過ちを犯してしまっている。これでは定数関数 λz.x が恒等関数になってしまう。今回もやはり、定数関数中の束縛変数の名前として z を選んでしまっていたばかりに問題が起こっており、明らかにまだ何かがおかしい。

項 s 中の自由変数が項 t 中に代入される際に単純に束縛されてしまう現象は、**変数捕獲**と呼ばれる。これを避けるためには、t の束縛変数の名前が s の自由変数の名前と異なっていることを保証する必要がある。これを正しく実現した代入操作は**捕獲回避代入**と呼ばれる。（単に「代入」といったときには、ほぼ常にこれのことを指す。）望む効果を得るには、ラムダ抽象の場合の第二の節に付帯条件を追加すればよい。

$$\begin{array}{lcll}
[\mathtt{x} \mapsto \mathtt{s}]\mathtt{x} & = & \mathtt{s} & \\
[\mathtt{x} \mapsto \mathtt{s}]\mathtt{y} & = & \mathtt{y} & \mathtt{y} \neq \mathtt{x}\text{ の場合} \\
[\mathtt{x} \mapsto \mathtt{s}](\lambda\mathtt{y.t_1}) & = & \begin{cases} \lambda\mathtt{y.\ t_1} & \mathtt{y} = \mathtt{x}\text{ の場合} \\ \lambda\mathtt{y.\ }[\mathtt{x} \mapsto \mathtt{s}]\mathtt{t_1} & \mathtt{y} \neq \mathtt{x}\text{ かつ }\mathtt{y} \notin FV(\mathtt{s})\text{ の場合} \end{cases} \\
[\mathtt{x} \mapsto \mathtt{s}](\mathtt{t_1\ t_2}) & = & ([\mathtt{x} \mapsto \mathtt{s}]\mathtt{t_1})\ ([\mathtt{x} \mapsto \mathtt{s}]\mathtt{t_2}) &
\end{array}$$

代入のこの定義は、**少なくとも代入が行われる際には**正しく行われるので、これでほとんどよい。今度の問題は、この修正によって代入が部分的な演算になってしまったことである。例えば、この新しい定義は $[\mathtt{x} \mapsto \mathtt{y\ z}](\lambda\mathtt{y.\ x\ y})$ に対しては何の結果も与えない。代入先の項の束縛変数 y は x と等しくはないが、(y z) に自由に現れるため、定義のいずれの節にも当てはまらないからである。

型システムやラムダ計算の文献ではしばしば、最後の問題に対する修正として、項を「束縛変数名の付け替えによる違いを無視して」扱う。（Church は、項中の束縛変数の名前を一貫して付け替える操作を**アルファ変換**と呼んだ。この用語法はいまだ一般的であるので、項を「アルファ変換による違いを無視して」扱うという言い方をすることもある。）

→（型無し）

構文			評価	$t \longrightarrow t'$
t ::=		項：	$\dfrac{t_1 \longrightarrow t_1'}{t_1\ t_2 \longrightarrow t_1'\ t_2}$	(E-APP1)
	x	変数		
	λx.t	ラムダ抽象	$\dfrac{t_2 \longrightarrow t_2'}{v_1\ t_2 \longrightarrow v_1\ t_2'}$	(E-APP2)
	t t	関数適用		
v ::=		値：		
	λx.t	ラムダ抽象値	$(\lambda x.t_{12})\ v_2 \longrightarrow [x \mapsto v_2]t_{12}$	(E-APPABS)

▲ 図 5-3. 型無しラムダ計算（λ）

慣習 5.3.4. 束縛変数の名前のみが異なる項は、任意の文脈で置き換え可能である。 □

これが実際に意味するのは、λ 束縛された変数の名前は、都合が良ければいつでも、（λ の本体で同じ変更を一貫して行うことで）別の名前へと変更できるということである。例えば、$[x \mapsto y\ z](\lambda y.\ x\ y)$ を計算したいときには、最初に $(\lambda y.\ x\ y)$ を例えば $(\lambda w.\ x\ w)$ へと書き換える。その後、$[x \mapsto y\ z](\lambda w.\ x\ w)$ を計算し、$(\lambda w.\ y\ z\ w)$ を得る。

この慣習はこの代入操作を「全域的も同然」にする。なぜなら、代入操作が未定義になる引数に適用しようとしていることに気づいたときには常に、必要に応じて変数名を変更し、付帯条件を満たすようにできるためである。実際、この慣習を採用することにより、代入の定義をさらに少しだけ簡潔に定式化できる。ラムダ抽象を扱う最初の節は取り除ける。なぜなら、（必要ならば名前の変更を行うことにより）常に束縛変数 y が x と s の自由変数のいずれとも異なることを仮定できるためである。これで代入の定義の最終形が得られる。

定義 5.3.5.［代入］

$$
\begin{array}{lcll}
[x \mapsto s]x & = & s & \\
[x \mapsto s]y & = & y & y \neq x \text{ の場合} \\
[x \mapsto s](\lambda y.t_1) & = & \lambda y.\ [x \mapsto s]t_1 & y \neq x \text{ かつ } y \notin FV(s) \text{ の場合} \\
[x \mapsto s](t_1\ t_2) & = & ([x \mapsto s]t_1)\ ([x \mapsto s]t_2) &
\end{array}
$$
□

操作的意味論

図 5-3 はラムダ項の操作的意味論の要約である。算術式の場合と比べて、値の集合が興味深いものになっている。（値呼びの）評価はラムダに到達すると評価を停止するため、値は任意のラムダ抽象であってよい。

評価関係は図の右列のものである。算術式の場合と同様に二種類の規則、すなわち**計算**規則 E-APPABS、および**合同**規則 E-APP1・E-APP2 がある。

メタ変数の選択が評価順の制御を補助していることに注意されたい。v_2 は値の上のみを動くため、規則 E-APPABS の左辺は引数部分が値である関数適用にしかマッチしない。同様に、規則 E-APP1 は関数部分が値でない関数適用に使われる。t_1 は任意の項にマッチするためである。ただし、前提条件によって t_1 の項が 1 ステップ実行可能であることが要求されている。一方、E-APP2

は、左辺が値**である**ようになるまで、すなわち値を表すメタ変数 v で束縛できるようになるまで、評価されない。これらを合わせると、関数適用 t_1 t_2 の評価順は以下の順に完全に定まる。まず、E-App1 を用いて t_1 を値に評価し、続いて E-App2 を用いて t_2 を値に評価する。最後に E-AppAbs で関数の適用そのものを行う。

演習 5.3.6. [★★] これらの規則を変更し、他の三種の評価戦略（完全ベータ簡約、正規順序、遅延評価）を表現するようにせよ。 □

純粋なラムダ計算では、値としてありうるのはラムダ抽象しかないことに注意されたい。そのため、E-App1 が t_1 を値に簡約することに成功した状態に到達したら、その値はラムダ抽象でしかありえない。これは、もちろん他の構成要素（例えばプリミティブなブール値）を言語に追加すると、ラムダ抽象以外の値の形式が導入されるので、成り立たなくなる。

演習 5.3.7. [★★ ↛] 演習 3.5.16 ではブール値と算術式の別の操作的意味論として、行き詰まり状態の項が特別な定数 wrong へと評価されるものを定義した。この意味論を**λNB** へと拡張せよ。 □

演習 5.3.8. [★★] 演習 3.5.17 は「大ステップ」スタイルでの算術式の評価を導入し、このスタイルでは基本評価関係が「項 t は最終結果 v へと評価される」というものだった。ラムダ項の評価規則を大ステップスタイルで形式化するにはどうすればよいかを示せ。 □

5.4 注記

型無しラムダ計算は Church とその同僚たちによって 1920 年代から 1930 年代にかけて作られた（Church [101]）。型無しラムダ計算のあらゆる側面を扱っている標準的な教科書は Barendregt [37] である。Hindley and Seldin [210] は包括性では劣るが、より読みやすい。"*Handbook of Theoretical Computer Science*" に収められた Barendregt の論文 [38] はコンパクトなサーベイである。ラムダ計算に関しては、関数型プログラミング言語の教科書（Abelson and Sussman [11]; Friedman et al. [163]; Peyton Jones and Lester [354] など）やプログラミング言語意味論の教科書（Schmidt [427]; Gunter [188]; Winskel [496]; Mitchell [322] など）の多くでも、解説されている。さまざまな種類のデータ構造をラムダ項に系統的に表現する手法は Böhm and Berarducci [50] で解説されている。

カリー化という名前にもかかわらず、Curry は発明者が自分であることを否定している。発明の功績は Schönfinkel [429] に帰されるのが通常であるが、その根底となるアイディアは Frege や Cantor といった 19 世紀の数学者にはよく知られたものであった。

実際、この体系には論理として使う以外の応用があるのかもしれない。
—*Alonzo Church, 1932*

第6章

項の名無し表現

本章では、純粋な型無しラムダ計算 λ（図 5-3）を扱う。関連する OCaml 実装は `fulluntyped` である。

前の章で我々は項を「束縛変数の名前の違いを除いて」扱った。すなわち、代入を可能にするためなど何らかの理由により束縛変数に対して新しい名前を与えることが好ましい場合、一般にどの時点でも名前を変えてよいという慣習を導入した。これは実質、束縛変数名の「綴り」は何でもよいということになる。この慣習は基本的な概念を議論したり証明をきれいに示したりするにはよいが、実装を行うためには、一つの項に対し一つの表現を選ばなければならない。特に、変数の出現をどのように表現するか決めなければならない。それには幾つかの方法がある。

(1) これまでしてきたように、変数を記号的に表現する。ただし束縛変数の名前を暗に変える慣習ではなく、捕獲を避けるために必要であれば、代入の際に束縛変数を明示的に「フレッシュな」名前に置き換える操作を行う。

(2) 同じく変数を記号的に表現する。ただし束縛変数の名前はすべて、互いに異なり、また、使う可能性のあるどの自由変数とも異なるという一般的な条件を導入する。この慣習（**Barendregt の慣習**とも呼ばれる）は我々の慣習より厳しく、任意の時点での「その場での」名前の付け替えを許容しない。しかしこの条件は代入（やベータ簡約）の下で安定ではない。なぜならば、代入操作は代入される項の複製を伴うので、代入結果が、同じ束縛変数名を持つ複数の λ 抽象を含むような項となる例も容易に構成できるからである。よって代入を含む評価ステップはいつも、不変条件を回復するための名前変更のステップを伴う必要があることになる。

(3) 名前を変更する必要がない、変数および項の「標準の」表現を考える。

(4) **明示的代入**などの仕組みを導入することで、代入そのものを避ける（Abadi, Cardelli, Curien, and Lévy [6]）。

(5) コンビネータに直接基づいた言語を使うことで、**変数**の使用を完全に避ける。例として、ラムダ計算の変種で、手続きによる抽象化に代えてコンビネータを基礎とする**コンビネータ論理**（Curry and Feys [128]; Barendregt [37]）や、Backus の関数型言語 FP（Backus [33]）がある。

どの仕組みもそれぞれ支持者がおり、どれを選ぶかはある程度好みの問題でもある（まじめにコンパイラを実装するにあたっては性能についても考慮の必要があるが、ここでは扱わない）。我々は三番目を選択する。経験によれば、そのほうが、より複雑な実装の幾つかを扱う際、扱いが容易になる。その主な理由は、この方法による実装が間違っている場合、微妙に失敗するのではなく壊滅的に失敗する傾向があるからである。そのため、比較的早く間違いを検出し修正できる。対照的に、名前付き変数を使った実装におけるバグは、忍び込んでから見つかるまでに数ヶ月ないし数年もかかることが知られている。以下の定式化では、よく知られた、Nicolas de Bruijn による技法 [134] を用いる。

6.1 項と文脈

de Bruijn のアイディアとは、読みやすさを犠牲にすれば、項の表現はもっと単純にできるというものであった。変数の出現を、名前を介して参照するのではなく、束縛子を**直接指す**ことによって項を表現するのである。これは、名前の付いた変数を自然数に置き換えることで達成できる。自然数 k は、「それを囲む k 番目の λ によって束縛される変数」を意味する。例えば、通常の項 `λx.x` は**名無し項** `λ.0` に対応する。`λx.λy. x (y x)` は、`λ.λ. 1 (0 1)` に対応する。名無し項は時として **de Bruijn 項**とも呼ばれ、その中の変数を表現している数は **de Bruijn インデックス**と呼ばれる[†1]。コンパイラを書く人たちはこれと同じ概念について「静的距離」という語を用いる。

演習 6.1.1. [★]　次のコンビネータそれぞれについて、対応する名無し項を書け。

```
c₀ = λs. λz. z;
c₂ = λs. λz. s (s z);
plus = λm. λn. λs. λz. m s (n s z);
fix = λf. (λx. f (λy. (x x) y)) (λx. f (λy. (x x) y));
foo = (λx. (λx. x)) (λx. x);
```

□

形式的には、名無し項の構文は通常の項の構文（定義 5.3.1）とほとんど同じように定義する。唯一の違いは、各項が自由変数を幾つ含みうるかについて丁寧な追跡が必要になることである。すなわち、自由変数を全く含まない項（**0 項**と呼ばれる）の集合、高々一つの自由変数を含む項（**1 項**と呼ばれる）の集合、などを区別する。

定義 6.1.2. [**項**]　$\mathcal{T}$ を、以下の条件を満たす集合の最小の族 $\{\mathcal{T}_0, \mathcal{T}_1, \mathcal{T}_2, \ldots\}$ とする。

(1)　$0 \leq \mathtt{k} < n$ ならば $\mathtt{k} \in \mathcal{T}_n$

(2)　$\mathtt{t}_1 \in \mathcal{T}_n$ かつ $n > 0$ ならば $\lambda.\mathtt{t}_1 \in \mathcal{T}_{n-1}$

(3)　$\mathtt{t}_1 \in \mathcal{T}_n$ かつ $\mathtt{t}_2 \in \mathcal{T}_n$ ならば $(\mathtt{t}_1\ \mathtt{t}_2) \in \mathcal{T}_n$

（注意として、これはほとんど普通の帰納的定義であるが、一つの集合を定義しているのではなく数で添字付けられた集合**族**を定義している点だけが異なる。）各 $\mathcal{T}_n$ の要素を n 項と呼ぶ。　□

$\mathcal{T}_n$ の要素は、**高々** n 種類の自由変数を持つ項であり、それらの変数には 0 から $n-1$ までの数が振られる。つまり、ある与えられた $\mathcal{T}_n$ の要素に、0 から $n-1$ までのすべての数が振られた自由変数が含まれる必要はなく、全く自由変数を含まなくてもよい。例えば閉じた項 t は、任意の n について $\mathcal{T}_n$ に属する。

（閉じた）通常の項が、それぞれ唯一の de Bruijn 表現を持つことに注意されたい。二つの通常の項を考えたとき、それらが束縛変数名の付け替えによる違いを無視して等しければ de Bruijn 表現も等しく、また逆も成り立つことにも注意されたい。

自由変数を含む項を扱うためには、**名前付け文脈**の仕組みが必要である。例として `λx. y x` を名無し項として表現することを考える。x の扱い方は既にわかっているが、y については束縛子が見つからず、したがってこの変数がどれだけ「遠く」なりうるのかわからず、どの数を割り振ればいいかわからない。解決法は、自由変数への de Bruijn インデックスの割り当て方（これを名前付け文脈と呼ぶ）をあらかじめ決めてしまい、自由変数に対して数を選ぶ必要が生じたときは一貫してそれに従うというものである。例として、次の名前付け文脈を用いることを考える。

[†1] 発音についての注意："de Bruijn" の発音に日本語で最も近いのは「ドブラウン」であり「デブルーイン」ではない。

$$\Gamma \;=\; \begin{array}{l} \mathtt{x} \mapsto 4 \\ \mathtt{y} \mapsto 3 \\ \mathtt{z} \mapsto 2 \\ \mathtt{a} \mapsto 1 \\ \mathtt{b} \mapsto 0 \end{array}$$

すると x (y z) は 4 (3 2) と表現される。λw. y w は λ. 4 0 と表現される。λw.λa.x は λ.λ.6 と表現される。

変数のインデックスは Γ に出現する順序で決まるので、名前付け文脈は列として簡潔に書ける。

定義 6.1.3. x_0 から x_n を、$\mathcal{V}$ から取ってきた変数名とする。名前付け文脈 $\Gamma = x_n, x_{n-1}, \ldots, x_1, x_0$ とは、各 x_i に de Bruijn インデックス i を割り振ったものである。列の中で右端にある変数にインデックス 0 を与えることとなる。これは、名前付き項を名無し項に変換するときに λ 束縛子を右から左へと数えるやり方と合致する。$dom(\Gamma)$ と書いて、Γ で扱われる変数名の集合 $\{x_n, \ldots, x_0\}$ を意味することにする。 □

演習 6.1.4. [★★★ ↛]　n 項の集合を定義 3.2.3 の様式で構成し直し、それが上で定義したものと等しいことを（命題 3.2.6 で行ったのと同様に）示せ。 □

演習 6.1.5. [推奨, ★★★]

(1) 名前付け文脈 Γ と通常の項 t （ただし $FV(\mathtt{t}) \subseteq dom(\Gamma)$ とする）を取り、対応する名無し項を返す関数 $removenames_\Gamma(\mathtt{t})$ を定義せよ。
(2) 名無し項 t と名前付け文脈 Γ を取り、通常の項を作る関数 $restorenames_\Gamma(\mathtt{t})$ を定義せよ。(t の中のラムダ抽象によって束縛されている変数に対して、名前を「作り出す」必要がある。Γ に含まれる名前が互いに異なるということと、変数名の集合 $\mathcal{V}$ が順序付けられているということは仮定してよい。この仮定により「$dom(\Gamma)$ に含まれないような最初の変数名を選ぶ」という表現が意味をなすようにできる。)

この一対の関数は次の二つの性質を持つようにせよ。

- 任意の名無し項 t について、$removenames_\Gamma(restorenames_\Gamma(\mathtt{t})) = \mathtt{t}$
- 任意の通常の項 t について、束縛変数名の付け替えによる違いを無視すると、$restorenames_\Gamma(removenames_\Gamma(\mathtt{t})) = \mathtt{t}$ □

厳密に言うと、「ある $\mathtt{t} \in \mathcal{T}$ について」という表現は意味をなさない。t が持ちうる自由変数の数は、常に指定されなければならない。しかし実際のところは、何らかの固定的な名前付け文脈 Γ を想定しているものである。そこで、記法を多少乱用し、$\mathtt{t} \in \mathcal{T}$ と書いて $\mathtt{t} \in \mathcal{T}_n$（ただし n は Γ の長さ）を意味することにする。

6.2 シフトと代入

次の仕事は名無し項に対する代入操作（$[\mathtt{k} \mapsto \mathtt{s}]\mathtt{t}$）を定義することである。このためには、「シフト」と呼ばれる補助的な操作が一つ要る。この操作は項に現れる自由変数のインデックスを付け替える。

代入が λ 抽象の中で行われる場合、例えば $[1 \mapsto \mathtt{s}](\lambda.2)$ （すなわち、外側の文脈において 1 が x のインデックスであると仮定したときの、$[\mathtt{x} \mapsto \mathtt{s}](\lambda \mathtt{y}.\mathtt{x})$）のような場合、代入が行われる段階での文脈は、元の文脈よりも 1 変数分長いものになる。s の中の自由変数のインデックスは、新し

い文脈において元の文脈と同じ名前を指すように、1 が加えられなければならない。ただしこの操作には注意が必要である。単に s の中のすべての変数についてインデックスを 1 シフトすることはできない。そのようなことをすれば、s の中の**束縛**変数までもシフトすることになってしまう。例として、s = 2 (λ.0)（すなわち、外側の文脈において 2 が z のインデックスであると仮定したときの、s = z (λw.w)）を考えると、2 はシフトしなければならないが、0 をシフトしてはならない。定義 6.2.1 のシフト関数は「打ち切り値」パラメータ c を取り、これによってどの変数がシフトされるべきかを制御する。これはまず 0 から始まり（すべての変数がシフトされることを意味する）、シフト関数が束縛子を通るたびに 1 が加えられてゆく。$\uparrow^d_c(\mathtt{t})$ を計算するときには、$\uparrow^d$ に元々渡された引数において、t は c 個の束縛子の内側にあったものだとわかっていることになる。よって t の中の任意の識別子 k で $\mathtt{k} < c$ となるものは元の引数の中で束縛されており、したがってシフトされてはならない。一方 $\mathtt{k} \geq c$ となるものは自由であり、したがってシフトされるべきである。

定義 6.2.1.［シフト］　項 t の、打ち切り値 c 以上の d 個シフトを $\uparrow^d_c(\mathtt{t})$ と書き、次のように定義する。

$$
\begin{array}{lcl}
\uparrow^d_c(\mathtt{k}) & = & \begin{cases} \mathtt{k} & k < c \text{ の場合} \\ \mathtt{k}+d & k \geq c \text{ の場合} \end{cases} \\
\uparrow^d_c(\lambda.\mathtt{t}_1) & = & \lambda.\ \uparrow^d_{c+1}(\mathtt{t}_1) \\
\uparrow^d_c(\mathtt{t}_1\ \mathtt{t}_2) & = & \uparrow^d_c(\mathtt{t}_1)\ \uparrow^d_c(\mathtt{t}_2)
\end{array}
$$

$\uparrow^d(\mathtt{t})$ は $\uparrow^d_0(\mathtt{t})$ を表すとする。　□

演習 6.2.2.［★］

(1) $\uparrow^2(\lambda.\lambda.\ 1\ (0\ 2))$ を計算せよ。
(2) $\uparrow^2(\lambda.\ 0\ 1\ (\lambda.\ 0\ 1\ 2))$ を計算せよ。　□

演習 6.2.3.［★★ ↛］　t が n 項であるとする。加えて $d < 0$ である場合には、t の自由変数はどれも $|d|$ 以上であるとする。このとき、$\uparrow^d_c(\mathtt{t})$ が $\max(n{+}d, 0)$ 項であることを示せ。　□

ここまでで代入操作 $[\mathtt{j} \mapsto \mathtt{s}]\mathtt{t}$ を定義する準備ができた。代入を行うとき、通常興味があるのは文脈中**最後**にある変数（すなわち、j = 0）についての代入である。これがベータ簡約の操作を定義するにあたり必要になるからである。しかし λ 抽象の項の内部で変数 0 への代入を行うためには、抽象の本体で 1 が振られている変数への代入が可能でなければならない。よって代入の定義は任意の変数について有効なものでなければならない。

定義 6.2.4.［代入］　項 t における、変数番号 j への項 s の代入を、$[\mathtt{j} \mapsto \mathtt{s}]\mathtt{t}$ と書き、次のように定義する。

$$
\begin{array}{lcl}
[\mathtt{j} \mapsto \mathtt{s}]\mathtt{k} & = & \begin{cases} \mathtt{s} & \mathtt{k} = \mathtt{j} \text{ の場合} \\ \mathtt{k} & \text{それ以外の場合} \end{cases} \\
[\mathtt{j} \mapsto \mathtt{s}](\lambda.\mathtt{t}_1) & = & \lambda.\ [\mathtt{j}{+}1 \mapsto \uparrow^1(\mathtt{s})]\mathtt{t}_1 \\
[\mathtt{j} \mapsto \mathtt{s}](\mathtt{t}_1\ \mathtt{t}_2) & = & ([\mathtt{j} \mapsto \mathtt{s}]\mathtt{t}_1)\ ([\mathtt{j} \mapsto \mathtt{s}]\mathtt{t}_2)
\end{array}
$$
□

演習 6.2.5.［★］　大域的文脈を Γ = a,b として以下の代入操作の例を名無し形式に変換し、上の定義に従って結果を計算せよ。その結果が、通常の項に対する代入（52 ページ）の元の定義と一致するか比べよ。

(1) [b ↦ a] (b (λx.λy.b))
(2) [b ↦ a (λz.a)] (b (λx.b))

(3) $[\mathtt{b} \mapsto \mathtt{a}]\,(\lambda\mathtt{b}.\ \mathtt{b}\ \mathtt{a})$
(4) $[\mathtt{b} \mapsto \mathtt{a}]\,(\lambda\mathtt{a}.\ \mathtt{b}\ \mathtt{a})$ □

演習 6.2.6. [★★ ↛] s と t が n 項であり、$\mathtt{j} \le n$ であるとするとき、$[\mathtt{j} \mapsto \mathtt{s}]\mathtt{t}$ が n 項であることを示せ。 □

演習 6.2.7. [★ ↛] 一枚の紙を手に取り、上で述べた代入とシフトについての定義を、見直さずに書いてみせよ。 □

演習 6.2.8. [推奨, ★★★] 名無し項についての代入の定義は、通常の項についての代入の定義[†2]と合致するべきであるが、(1) その対応を厳密に正当化するにはどのような定理が証明される必要があるかを考え、(2) それを証明せよ。 □

6.3 評価

名無し項についての評価関係を定義するためには、ベータ簡約の規則だけ（変数名が関係してくるのはここだけである）を変え、名無しでの代入操作を用いるようにすればよい。

ほんの少しだけ問題となるのは、簡約基を簡約することで、束縛変数が「消費されて」しまうことである。$((\lambda\mathtt{x}.\mathtt{t}_{12})\ \mathtt{v}_2)$ を $[\mathtt{x} \mapsto \mathtt{v}_2]\mathtt{t}_{12}$ と簡約するとき、その過程で束縛変数 x は消えてしまう。よって代入結果の中の変数に対しては再度番号を振り、x が文脈から消えたことを反映するようにしなければならない。例を挙げると、

$$(\lambda.1\ 0\ 2)\ (\lambda.0) \longrightarrow 0\ (\lambda.0)\ 1 \qquad (1\ (\lambda.0)\ 2\ \text{ではなく})$$

のようになる。また、$\mathtt{t}_{12}$ は $\mathtt{v}_2$ よりも大きな文脈[†3]で定義されているため、$\mathtt{v}_2$ の中の変数については、$\mathtt{t}_{12}$ への代入が行われる前に 1 を加えるシフトを行う必要がある。これらの点を考慮した上で、ベータ簡約規則は次のようになる。

$$(\lambda.\mathtt{t}_{12})\ \mathtt{v}_2 \quad\longrightarrow\quad \uparrow^{-1}([0 \mapsto \uparrow^{1}(\mathtt{v}_2)]\mathtt{t}_{12}) \qquad \text{(E-AppAbs)}$$

他の規則は以前定義したもの（図 5-3）と同じである。

演習 6.3.1. [★] この規則にある負のシフトについて、負のインデックスを含む正しくない形の項が作られるおそれがあるかどうか考えよ。 □

演習 6.3.2. [★★★] de Bruijn の元々の論文には、項の名無し表現について、二つの異なる提案が含まれていた。一つはここで述べた de Bruijn **インデックス**であり、ラムダ束縛子に対し「内から外へ」番号を振る。もう一つは **de Bruijn レベル**であり、束縛子に対し「外から内へ」番号を振る。例えば、項 $\lambda\mathtt{x}.\ (\lambda\mathtt{y}.\ \mathtt{x}\ \mathtt{y})\ \mathtt{x}$ を de Bruijn インデックスを用いて表現すると $\lambda.\ (\lambda.\ 1\,0)\ 0$ となり、de Bruijn レベルを用いて表現すると $\lambda.\ (\lambda.\ 0\,1)\ 0$ となる。de Bruijn レベルを正確に定義し、インデックスとレベルによる項の表現が同型であること（すなわち、互いに一方から他方を唯一の形に復元できること）を示せ。 □

[†2] （訳注）定義 5.3.5 のこと。
[†3] （訳注）項 $\mathtt{t}_{12}$ を囲む λ の数が $\mathtt{v}_2$ より多いという意。

第 7 章

ラムダ計算のML実装

本章のほとんどでは純粋な型無しラムダ計算を扱う（図 5-3）。対応する実装は untyped である。また、実装 fulluntyped は数やブール値といった拡張を含んでいる。

この章では、第 4 章の算術式のインタプリタおよび、第 6 章の変数束縛および代入の扱いに基づいて、型無しラムダ計算のインタプリタを構築する。

型無しのラムダ項の実行可能な評価器は、前述の定義をそのまま OCaml に変換することで得られる。第 4 章と同様に、核となるアルゴリズムだけを示し、字句解析、構文解析、表示といった事項については扱わない。

7.1 項と文脈

項の抽象構文木を表すデータ型は、定義 6.1.2 を直接翻訳することで得られる。

```
type term =
    TmVar of int
  | TmAbs of term
  | TmApp of term * term
```

変数は数、すなわち de Bruijn インデックスで表現される。ラムダ抽象は、ラムダ抽象の本体である部分項のみを保持する。関数適用は適用される二つの部分項[†1]を保持する。

ただし、我々の実装で実際に使われている定義は、もう少しだけ情報を保持している。第一に、前と同様に、各項に対して info 型の要素を注記として付け加え、その項が元々あったファイル位置を記録すると便利である。エラー表示ルーチンはユーザを（あるいは、ユーザのテキストエディタも、自動的に）エラーが起こった地点へ正確に誘導できる。

```
type term =
    TmVar of info * int
  | TmAbs of info * term
  | TmApp of info * term * term
```

第二に、デバッグのため、各変数ノードに対してもう一つ数を付け加え、一貫性検査を行うと有用である。付け加えた数は、その変数が現れる文脈**全体の長さ**を常に保持するようにする。

```
type term =
    TmVar of info * int * int
  | TmAbs of info * term
  | TmApp of info * term * term
```

変数を表示する際には、この数が現在の文脈の実際のサイズと一致するか確かめる。もし一致しない場合は、シフト演算をどこかで忘れたということである。

最後の改良も表示に関するものである。項は内部的には de Bruijn インデックスを使って表現されているが、これは明らかにユーザに提示するべき形ではない。構文解析時に通常の表現を名無し項へと変換し、表示時には通常の形式に戻すべきである。これは別に難しくはないものの、完全に

[†1] （訳注）関数部分と引数部分のこと。

素朴な方法は望ましくない。例えば、変数名として完全にフレッシュな名前を生成するべきではない。そのようにしてしまうと、表示される項中の束縛変数名が、元のプログラム中の名前と全く無関係になってしまうためである。この問題は、各ラムダ抽象に束縛変数名のヒントとなる文字列を注記することで解決できる。

```
type term =
    TmVar of info * int * int
  | TmAbs of info * string * term
  | TmApp of info * term * term
```

項に対する基本操作（特に代入）は、これらの文字列に対して特別な処理は行わない。これら文字列はそのままの形で受け渡され、名前の衝突や捕獲などの検査も行われない。表示ルーチンにおいて束縛変数用にフレッシュな名前を生成する必要が生じたら、まずヒントの文字列を使うことを試み、もしその文字列が現在の文脈において既に使われている名前と衝突するようならば、同様の名前を使うことを試みる。すなわち、現在使われている名前と衝突しない名前が見つかるまで、プライム記号（'）を追加していく。これにより、ユーザの期待するものと（幾つかのプライム記号による違いを除いて）同様の項を表示できる。

表示ルーチン自体は以下のようになる。

```
let rec printtm ctx t = match t with
    TmAbs(fi,x,t1) →
      let (ctx',x') = pickfreshname ctx x in
      pr "(lambda "; pr x'; pr ". "; printtm ctx' t1; pr ")"
  | TmApp(fi, t1, t2) →
      pr "("; printtm ctx t1; pr " "; printtm ctx t2; pr ")"
  | TmVar(fi,x,n) →
      if ctxlength ctx = n then
        pr (index2name fi ctx x)
      else
        pr "[bad index]"
```

ここで用いた context（文脈）型

```
type context = (string * binding) list
```

は、文字列とそれに関連付けられた binding（束縛）からなる組のリストである。差し当たり、束縛自体は

```
type binding = NameBind
```

という全く自明な定義であり、有用な情報を持たない。後に第 10 章では、binding 型に節を追加し、変数に関連付けられた型の仮定や、他の同種の情報を記録する。

表示関数は低水準の関数も幾つか用いる。pr は文字列を標準出力ストリームに送る。ctxlength は文脈の長さを返す。index2name は変数のインデックスから名前の文字列を引く。最も面白いのは pickfreshname で、これは文脈 ctx とヒントの文字列 x を受け取る。そして、x に似ているが ctx に含まれていない x' を探し、その x' を ctx に加えて新たな文脈 ctx' を作り、ctx' と x' を組として返す。

この本のウェブサイトにある untyped の実装にある実際の表示関数は、もう少し複雑になっており、上述に加え二つの問題を取り扱っている。第一に、関数適用は左に結合し、ラムダ抽象の本体は可能な限り右まで含めて解釈するという慣習に従うことで、できるだけ多くの括弧を省略する。第二に、ウェブサイトにある実装は下位の**整形表示**モジュール（OCaml の Format ライブラリ）への整形命令を生成するようになっており、このモジュールが改行や字下げの判断を行う。

7.2 シフトと代入

シフトの定義（定義 6.2.1）は、ほぼ一対一に記号を置き換えるだけで、OCaml へと翻訳できる。

```
let termShift d t =
  let rec walk c t = match t with
    TmVar(fi,x,n) → if x>=c then TmVar(fi,x+d,n+d)
                    else TmVar(fi,x,n+d)
  | TmAbs(fi,x,t1) → TmAbs(fi, x, walk (c+1) t1)
  | TmApp(fi,t1,t2) → TmApp(fi, walk c t1, walk c t2)
  in walk 0 t
```

内部のシフト $\uparrow^d_c$(t) はここでは、内部関数の呼び出し walk c t で表現している。d は常に変わらないので、walk の呼び出しごとに受け渡す必要はない。変数の場合の walk の定義で d が必要になったときには、単に外側の d の束縛を利用している。トップレベルのシフト $\uparrow^d$(t) は termShift d t で表現している。（termShift 自体は再帰を表す rec が付いていないことに注意されたい。これは termShift は walk を一度呼ぶだけだからである。）

同様に、代入関数も定義 6.2.4 からほぼ直接得られる。

```
let termSubst j s t =
  let rec walk c t = match t with
    TmVar(fi,x,n) → if x=j+c then termShift c s else TmVar(fi,x,n)
  | TmAbs(fi,x,t1) → TmAbs(fi, x, walk (c+1) t1)
  | TmApp(fi,t1,t2) → TmApp(fi, walk c t1, walk c t2)
  in walk 0 t
```

項 t 中の j 番の変数への項 s の代入 [j ↦ s]t は、ここでは termSubst j s t となる。代入の元々の定義との唯一の違いは、元々の定義では束縛子をくぐるごとに +1 ずつ s をシフトしていたのに対して、ここでは TmVar の場合に s のシフトをまとめて一回で済ませているという点である。その結果として、引数 j は walk の呼び出しごとに変化しないため、内部定義から省略できる。

読者は termShift の定義と termSubst の定義が非常によく似ていることに気づくかもしれない。違いは変数に至ったときに取る動作だけである。この本のウェブサイトから入手できる untyped の実装は、この観察を利用することによって、シフト操作と代入操作の双方を、より一般化された tmmap と呼ばれる関数の特殊な場合として表現している。項 t と関数 onvar に対し、tmmap onvar t の結果は、項 t と同じ形の項であって、各変数をその変数に対して onvar を呼び出した結果で置き換えたものである。この記法上の工夫を用いることで、大きな計算体系における面倒な繰り返しのかなりの部分を避けられる。この点については 25.2 節でより詳細に述べる。

ラムダ計算の操作的意味論では、代入が使われる唯一の箇所はベータ簡約規則である。既に書いたように、この規則は実際には幾つかの操作を行う。束縛変数を置き換える項をまず最初に +1 だけシフトする。次に代入を行う。最後にその束縛変数がなくなった事実に対応するため、結果全体を −1 だけシフトする。以下の定義はこの一連のステップを一つにまとめたものである。

```
let termSubstTop s t =
  termShift (-1) (termSubst 0 (termShift 1 s) t)
```

7.3 評価

第 4 章と同じく、評価関数は補助述語 isval を用いる。

```
let rec isval ctx t = match t with
    TmAbs(_,_,_) → true
  | _ → false
```

1 ステップの評価関数は、項と一緒に文脈 ctx を渡す点を除いて、評価規則をそのまま写したものになっている。この引数 ctx は現段階の eval1 関数では使っていないが、後により複雑な評価器の一部で必要となる。

```
let rec eval1 ctx t = match t with
    TmApp(fi,TmAbs(_,x,t12),v2) when isval ctx v2 →
      termSubstTop v2 t12
  | TmApp(fi,v1,t2) when isval ctx v1 →
      let t2' = eval1 ctx t2 in
      TmApp(fi, v1, t2')
  | TmApp(fi,t1,t2) →
      let t1' = eval1 ctx t1 in
      TmApp(fi, t1', t2)
  | _ →
      raise NoRuleApplies
```

多ステップの評価関数も、ctx 引数を除いて、前と同じである。

```
let rec eval ctx t =
  try let t' = eval1 ctx t
      in eval ctx t'
  with NoRuleApplies → t
```

演習 7.3.1. [推奨, ★★★ ↛]　この実装を演習 5.3.8 で導入した「大ステップ」スタイルを用いるように変更せよ。 □

7.4 注記

この章で示した代入の取り扱いは、この本の目的には十分であるものの、代入という話題に関する最終的結論には程遠い。特に、我々の評価器のベータ簡約規則は、引数の値を関数本体中の束縛変数へ「先行して」代入してしまっている。関数型言語のインタプリタ（やコンパイラ）で、単純さではなくスピードを重視してチューニングされているものは、異なる戦略を用いている。すなわち、実際に代入を行う代わりに、単に束縛変数名と引数値との対応を「**環境**」と呼ばれる補助的なデータ構造に記録し、評価する項と共に環境を持ち運ぶ。そして、変数に到達した時点ではじめて、その変数の値を現在の環境から調べる。この戦略は、環境を一種の**明示的代入**とみなすことでモデル化できる。すなわち、代入メカニズムをメタ言語から対象言語に移し、代入を項の外部からの操作ではなく、評価器によって操作される**構文**の一部とするのである。明示的代入を最初に研究したのは Abadi, Cardelli, Curien, and Lévy [6] であり、それ以来活発な研究領域となっている。

実装したからといって、理解したとは限らない。　*—Brian Cantwell Smith*

第II部

単純型

型付き算術式

本章で扱う体系はブール値と数の型付き計算（図 8-2）である。対応する OCaml 実装は tyarith である。

第 3 章では、ブール式と算術式の単純な言語を用いて、構文と評価の厳密な記述をするための基本的な道具立てを導入した。本章では、この単純な言語に立ち返り、静的な型で強化することを考える。第 3 章の言語と同様、本章の型システムもそれ自体はほぼ自明だが、本書を通じて繰り返し扱う概念を導入するための基礎を与える。

8.1 型

算術式の構文を思い出そう。

t ::=	項：
true	定数真
false	定数偽
if t then t else t	条件式
0	定数ゼロ
succ t	後者値
pred t	前者値
iszero t	ゼロ判定

第 3 章では、項を評価すると結果は値

v ::=	値：
true	真
false	偽
nv	数値

nv ::=	数値：
0	ゼロ
succ nv	後者値

になるか、または、ある段階で pred false のような、どの評価規則も適用できない項に辿り着いて**行き詰まり状態**になるかのどちらかであるということを見た。

行き詰まり状態の項は無意味または間違ったプログラムに対応する。したがって、項を実際に評価せずに、その項の評価が決して行き詰まり状態になら**ない**ことをいいたい。そのために、数値に評価される項（pred、succ、iszero の引数にできるのはこれだけ）と、ブール値に評価される項（条件式の条件部分となれるのはこれだけ）とを区別できる必要がある。項をこのように分類するために、二つの**型** Nat と Bool を導入する。メタ変数 S、T、U などはこの本を通じて型の上を動くものとする。

「項 t が型 T を持つ」（または「t が T に属する」、「t は T の要素である」）という表現を用いた場合、t を評価した結果が「明らかに」適切な形の値になることを意味する。ここで「明

𝔹（型付き） **B**（図 3-1）の拡張

新しい構文形式

`T`	::=	型：
	`Bool`	ブール値型

新しい型付け規則 $\boxed{\texttt{t : T}}$

$$\texttt{true : Bool} \qquad \text{(T-TRUE)}$$

$$\texttt{false : Bool} \qquad \text{(T-FALSE)}$$

$$\frac{\texttt{t}_1\texttt{ : Bool} \qquad \texttt{t}_2\texttt{ : T} \qquad \texttt{t}_3\texttt{ : T}}{\texttt{if t}_1\texttt{ then t}_2\texttt{ else t}_3\texttt{ : T}} \qquad \text{(T-IF)}$$

▲ 図 8-1. ブール値（**B**）のための型付け規則

𝔹 ℕ（型付き） **NB**（図 3-2）と図 8-1 の拡張

新しい構文形式

`T`	::= ...	型：
	`Nat`	自然数型

新しい型付け規則 $\boxed{\texttt{t : T}}$

$$\texttt{0 : Nat} \qquad \text{(T-ZERO)}$$

$$\frac{\texttt{t}_1\texttt{ : Nat}}{\texttt{succ t}_1\texttt{ : Nat}} \qquad \text{(T-SUCC)}$$

$$\frac{\texttt{t}_1\texttt{ : Nat}}{\texttt{pred t}_1\texttt{ : Nat}} \qquad \text{(T-PRED)}$$

$$\frac{\texttt{t}_1\texttt{ : Nat}}{\texttt{iszero t}_1\texttt{ : Bool}} \qquad \text{(T-ISZERO)}$$

▲ 図 8-2. 数（**NB**）のための型付け規則

らかに」といっているのは、**静的に**、つまり `t` を評価することなくわかるということである。例えば項 `if true then false else true` は `Bool` 型を持ち、`pred (succ (pred (succ 0)))` は `Nat` 型を持つ。しかし、項の型の分析は**保守的**であり、静的な情報しか用いない。これはつまり `if (iszero 0) then 0 else false` や `if true then 0 else false` のような項が、実際には行き詰まり状態にはならないにもかかわらず、どんな型も持てないということを意味する。

8.2 型付け関係

算術式のための型付け関係は、「`t : T`」と書き[†1]、項に型を割り当てる推論規則の集合によって定義される。推論規則の集合を図 8-1 と図 8-2 に要約する。第 3 章と同じように、ブール値のための規則と数のための規則を二つの異なる図として示している。これは、後で別々に参照したいことがあるからである。

図 8-1 の規則 T-TRUE と T-FALSE はブール定数 `true` と `false` に `Bool` 型を割り当てている。T-IF は条件式の部分式の型に基づいて条件式全体の型を割り当てている。条件部分 $\texttt{t}_1$ はブール値に評価されなければならず、$\texttt{t}_2$ と $\texttt{t}_3$ は**同じ**型の値に評価されなければならない。同じメタ変数 `T` を二回使用することで、`if` の結果は `then` 節および `else` 節の型で、かつこの型はどんな型にも（`Nat`

[†1] ：の代わりに、しばしば記号 ∈ を用いる。

やBool、またはより興味深い型のある計算体系ではその他の型にも）なりうる、という制約を表現している。

図 8-2 の、数に関する規則も同様の形をしている。T-Zero は Nat 型を定数 0 に割り当てている。T-Succ は t_1 が Nat 型であるような succ t_1 に Nat 型を割り当てている。同様に、T-Pred と T-IsZero も、pred は引数が Nat 型であるときに Nat 型となり、iszero は引数が Nat 型のときに Bool 型となることを述べている。

定義 8.2.1. 形式的には、算術式のための**型付け関係**は図 8-1 と図 8-2 における規則のすべてのインスタンスを満たす、項と型の最小の二項関係である。項 t に対して、ある型 T が存在して t : T となるとき、t は**型付け可能**（または**正しく型付けされている**）という。 □

型付け関係に関する推論をするとき、「succ t_1 という形の項が何らかの型を持つならば、それは Nat 型である」のような言い方をすることがよくある。次の補題はこのような形の基本的な言明をまとめたものであり、各項目は対応する型付け規則の形から直ちに導かれる。

補題 8.2.2. ［**型付け関係の逆転**］

(1) true : R ならば R = Bool である。
(2) false : R ならば R = Bool である。
(3) if t_1 then t_2 else t_3 : R ならば t_1 : Bool、t_2 : R かつ t_3 : R である。
(4) 0 : R ならば R = Nat である。
(5) succ t_1 : R ならば R = Nat かつ t_1 : Nat である。
(6) pred t_1 : R ならば R = Nat かつ t_1 : Nat である。
(7) iszero t_1 : R ならば R = Bool かつ t_1 : Nat である。 □

［証明］ 型付け関係の定義より直ちに成り立つ。 □

逆転補題は、型付け関係のための**生成補題**と呼ばれることもある。なぜなら、与えられた妥当な型付け判断式に対して、その証明がどのように生成されたかを示すからである。逆転補題から項の型を計算する再帰アルゴリズムを直接得られる。なぜなら、逆転補題は各構文形式の項の型（もし存在するのならば）を、部分項の型からどのように計算するかを述べているためである。この点について第 9 章でもう一度詳細に振り返る。

演習 8.2.3. ［★ ↛］ 正しく型付けされた項のすべての部分項は正しく型付けされていることを証明せよ。 □

3.5 節では評価導出の概念を導入した。同様に**型付け導出**は型付け規則のインスタンスの木である。 型付け関係に含まれる二つ組 (t, T) は t : T を結論とする型付け導出によって正当化される。例えば次は「if iszero 0 then 0 else pred 0 : Nat」の型付け判断式のための導出木である。

$$\dfrac{\dfrac{\dfrac{}{\mathtt{0 : Nat}}\ \textsc{T-Zero}}{\mathtt{iszero\ 0 : Bool}}\ \textsc{T-IsZero} \qquad \dfrac{}{\mathtt{0 : Nat}}\ \textsc{T-Zero} \qquad \dfrac{\dfrac{}{\mathtt{0 : Nat}}\ \textsc{T-Zero}}{\mathtt{pred\ 0 : Nat}}\ \textsc{T-Pred}}{\mathtt{if\ iszero\ 0\ then\ 0\ else\ pred\ 0 : Nat}}\ \textsc{T-If}$$

言い換えれば、**型付け判断式**はプログラムの型に関する形式的な主張である。**型付け規則**は型付け判断式の間の含意である。**型付け導出**は型付け規則に基づいた演繹である。

定理 8.2.4. ［**型の一意性**］ 各項 t は高々一つの型を持つ。つまり、もし t が型付け可能ならばそ

の型は一意である。さらに、図 8-1 と図 8-2 の推論規則から構築される型付け導出はたった一つだけである。 □

[証明] t に関する単純な構造的帰納法で、それぞれの場合に逆転補題の適切な節（と帰納法の仮定）を適用すればよい。 □

この章で扱う単純な型システムでは、すべての項は（何らかの型を持つならば）型を一つだけ持ち、さらにこのことを示す導出木も常に一つだけである。後に（例えば第 15 章で部分型付けを持つ型システムに取り掛かると）、両方の性質は緩められる。そこでは一つの項は複数の型を持つ可能性があり、与えられた項が与えられた型を持つことの型付け判断式の導出方法も一般的には複数存在しうる。

ちょうど評価関係の性質が評価導出に関する帰納法を使って証明されるように、型付け関係の性質はしばしば導出木に関する帰納法によって証明される。次の節から型付け導出に関する帰納法の多くの例を扱う。

8.3 安全性 = 進行 + 保存

この型システムでも他のどんな型システムでも、最も基本的な性質は**安全性**である（**健全性**とも呼ばれる）。これは、正しく型付けされた項は「おかしくなる」ことがないということである。項がおかしくなるとはどういうことか、を形式化する方法は既に決めた。それは、所定の最終的な値ではないが、次に適用する評価規則もない「行き詰まり状態」（定義 3.5.15）に到達するという意味だった。よって我々が知りたいことは、正しく型付けされた項は行き詰まり状態にならないということである。我々はこのことをよく知られた二つのステップである、**進行**定理と**保存**定理によって証明する[†2]。

- **進行**とは、正しく型付けされた項は行き詰まり状態ではない（値であるか、または評価規則によって評価を進められる）ということ。
- **保存**とは、正しく型付けされた項が評価できるならば、評価後の項もまた正しく型付けされている[†3]ということ。

これらの性質を合わせると、正しく型付けされた項は評価において行き詰まり状態になりえないとわかる。

進行定理の証明のために、Bool 型と Nat 型の**標準形**（すなわち、それらの型を持つ正しく型付けされた値）が取りうる形について、幾つかの事実を記しておくと便利である。

補題 8.3.1. [標準形]

(1) v が Bool 型の値ならば、v は true または false である。
(2) v が Nat 型の値ならば、v は図 3-2 の文法による数値である。 □

[証明] (1) について、図 3-1 と図 3-2 の文法によればこの言語における値は次の四つの形を持つ。true、false、0、succ nv（ただし nv は数値）である。最初の二つの場合は所望の結論を直ちに満たす。v が Bool 型であり、逆転補題（補題 8.2.2）の (4) と (5) により 0 と succ nv が Nat 型のみを

[†2] 「安全性とは進行および保存のことである」という（標準形補題を用いた）スローガンは、Harper によって述べられた。そしてその変種は Wright and Felleisen [500] で提案された。
[†3] 我々がこれから考えるほとんどの型システムでは、評価は正しく型付けできるかどうかだけを保存するのではなく、その項の型自体も保存する。しかし、型が評価の過程で変わりうる型システムもある。例えば部分型付けを持つ体系（第 15 章）では、型は評価の過程でより小さくなる（多くの情報を持つようになる）ことがある。

持つことができ、Bool 型を持たないので、残りの二つにはなりえない。(2) についても同様である。 □

定理 8.3.2.［進行］ t を正しく型付けされた項（つまりある T について t : T となる）と仮定する。このとき t は値であるか、またはある t′ が存在して t ⟶ t′ となる。 □

［証明］ t : T の導出に関する帰納法による。T-TRUE、T-FALSE、T-ZERO の場合は t が値であることから成立する。それ以外の場合に関して、以下で議論する。

T-IF の場合　t = if t_1 then t_2 else t_3
　　t_1 : Bool　t_2 : T　t_3 : T

帰納法の仮定より、t_1 は値であるか、または t_1' が存在して $t_1 \longrightarrow t_1'$ を満たす。t_1 が値ならば、標準形補題（補題 8.3.1）より true または false でなければならず、その場合 E-IFTRUE か E-IFFALSE が t に適用される。一方 $t_1 \longrightarrow t_1'$ ならば、E-IF により t ⟶ if t_1' then t_2 else t_3 となる。

T-SUCC の場合　t = succ t_1　t_1 : Nat

帰納法の仮定より、t_1 は値であるか、または t_1' が存在して $t_1 \longrightarrow t_1'$ を満たす。t_1 が値ならば、標準形補題により数値でなければならず、その場合 t も値である。一方 $t_1 \longrightarrow t_1'$ ならば、E-SUCC により succ t_1 ⟶ succ t_1' となる。

T-PRED の場合　t = pred t_1　t_1 : Nat

帰納法の仮定より、t_1 は値であるか、または t_1' が存在して $t_1 \longrightarrow t_1'$ を満たす。t_1 が値ならば、標準形補題により数値でなければならず、つまり 0 またはある nv_1 を使って succ nv_1 となり、規則 E-PREDZERO または E-PREDSUCC を t に適用できる。一方 $t_1 \longrightarrow t_1'$ ならば、E-PRED によって pred t_1 ⟶ pred t_1' となる。

T-ISZERO の場合　t = iszero t_1　t_1 : Nat

同様。 □

型が評価によって保存されることの証明も、この体系においてはかなり単純である。

定理 8.3.3.［保存］ t : T かつ t ⟶ t′ ならば、t′ : T となる。 □

［証明］ t : T の導出に関する帰納法による。帰納法の各ステップにおいて、すべての部分導出に関して所望の性質が成り立つ（つまり、s : S が部分導出によって証明されるときはいつでも、s : S かつ s ⟶ s′ ならば、s′ : S が成り立つ）と仮定する。そして、導出の最後の規則についての場合分けにより証明を進める。（場合分けの一部のみを証明するが、他の場合も同様である。）

T-TRUE の場合　t = true　T = Bool

導出の最後の規則が T-TRUE ならば、その規則の形から t が定数 true でなければならず、T が Bool であることがわかる。しかし t は値なのでどんな t′ に対しても t ⟶ t′ とならず、定理の要求は自明に満たされる。

T-IF の場合　t = if t_1 then t_2 else t_3　t_1 : Bool　t_2 : T　t_3 : T

導出の最後の規則が T-IF ならば、その規則の形から t はある t_1、t_2、t_3 に対して if t_1 then t_2 else t_3 という形でなければならない。さらに t_1 : Bool と t_2 : T と t_3 : T をそれぞれ結論とする部分導出があることもわかる。ここで（図 3-1 において）左側に if が現れる評価規則を見ると、t ⟶ t′ を導出できる規則は三つ、E-IFTRUE と E-IFFALSE と E-IF があるとわかる。それぞれの場合について別々に考える（E-IFFALSE の場合は E-IFTRUE と同様のため省略する）。

E-IfTrue の場合　t_1 = true　　$t' = t_2$

$t \longrightarrow t'$ が E-IfTrue を使って導出されたならば、この規則の形から t_1 は true であり、結果の項 t' は二つ目の部分式 t_2 となることがわかる。このことより（場合分け T-If のときの仮定により）t_2 : T であることがわかるので、所要の結果となる。

E-If の場合　$t_1 \longrightarrow t_1'$　　t' = if t_1' then t_2 else t_3

場合分け T-If での仮定から、結論が t_1 : Bool である、元の型付け導出の部分導出が得られる。帰納法の仮定をこの部分導出に適用して t_1' : Bool を得る。この事実と（場合分け T-If での仮定である）t_2 : T と t_3 : T を合わせると規則 T-If が適用でき、if t_1' then t_2 else t_3 : T となり、つまり t' : T が成り立つ。

T-Zero の場合　t = 0　　T = Nat

（上の T-True の議論と同じように）評価は起こりえない。

T-Succ の場合　t = succ t_1　　T = Nat　　t_1 : Nat

図 3-2 の評価規則を見ると、$t \longrightarrow t'$ を導くために使える規則は E-Succ 一つだけである。この規則の形から $t_1 \longrightarrow t_1'$ がわかる。t_1 : Nat であることもわかるので、帰納法の仮定より t_1' : Nat を得る。よって T-Succ を適用して succ t_1' : Nat となり、すなわち t' : T が成り立つ。　□

演習 8.3.4. [★★ ↛]　型付け導出ではなく評価導出に関する帰納法を用いてこの証明を再構築せよ。　□

保存定理はしばしば**主部簡約**（または **主部評価**）と呼ばれる。直観的に、型付け判断式 t : T は「t は型 T を持つ」という一文として考えられる。項 t はこの文の主語（主部）であり、主部簡約の性質は文の正しさが主部の簡約の下で保存されるということをいっている。

型の一意性に関してはそれを満たす型システムとそうでないものがある。しかし進行と保存は、我々が考えるすべての型システムで求められる基本的な性質である[†4]。

演習 8.3.5. [★]　評価規則 E-PredZero（図 3-2）は少し直観に反している。0 の前者値は 0 であるというより未定義であるほうが納得がいくと感じるかもしれない。この規則を 1 ステップ評価の定義から取り除くことによってこのことを簡単に達成できるだろうか。　□

演習 8.3.6. [推奨, ★★]　主部簡約の性質を見ていると、反対の性質（主部**展開**）も成り立つだろうかと思うことは妥当である。$t \longrightarrow t'$ かつ t' : T ならば t : T は常に成り立つだろうか。成り立つならば証明せよ。成り立たないならば反例を挙げよ。　□

演習 8.3.7. [推奨, ★★]　評価関係が演習 3.5.17 の大ステップスタイルで定義されていると仮定せよ。このとき直観的な型安全性はどのように形式化すべきか。　□

演習 8.3.8. [推奨, ★★]　演習 3.5.16 のように無意味な項を wrong という明示的な状態へ簡約する規則を追加した評価関係を仮定せよ。ここで型安全性はどのように形式化すべきか。　□

型無しの世界から型付きの世界への道は、何度も、多くの異なる分野で、おおよそ同じ理由により歩まれてきた。　—*Luca Cardelli and Peter Wegner, 1985* [88]

[†4] これらの性質を満たさないにもかかわらず、型安全と考えられる言語は**存在する**。例えば Java の操作的意味論を小ステップスタイルで形式化すると（Flatt, Krishnamurthi, and Felleisen [158]; Igarashi, Pierce, and Wadler [231]）、ここで与えた形の型保存は失敗する（詳細は第 19 章を見よ）。しかし、例えば大ステップによる意味論なら失敗はなくなるので、言語自身の欠陥ではなく、形式化の生み出した欠陥と考えるべきである。

単純型付きラムダ計算

本章で扱う体系はブール値（図 8-1）を含む単純型付きラムダ計算（図 9-1）である。対応する OCaml 実装は fullsimple である。

本章では、以降で扱う型付き言語の中でも、最も初歩的な言語である、Church [100] と Curry [128] の単純型付きラムダ計算を導入する。

9.1 関数型[†1]

第 8 章では、算術式の単純な静的型システムを二つの型と共に導入した。一つは Bool で、評価によりブール値となる項が分類される。もう一つは Nat で、評価により数となる項が分類される。どちらの型にも属さず型付けできない項には、評価すると行き詰まり状態に到達するすべての項（例えば if 0 then 1 else 2）が含まれる。また、実際には評価すると正常に振る舞う項であっても、ここでの静的な分類が保守的であるため、どちらの型にも分類できないような項（例えば if true then 0 else false）も含まれる。

ブール値（簡単のため、本章では数を無視する）と純粋なラムダ計算のプリミティブとを組み合わせた言語に対し、同様の型システムを構築したいとしよう。すなわち、変数、ラムダ抽象、関数適用のための、以下を満たすような型付け規則を導入したい。(a) 型安全性を維持する。つまり、型保存定理（定理 8.3.3）と進行定理（定理 8.3.2）を満たす。さらに (b) 保守的すぎない、つまり、書きたいプログラムのほとんどに型を割り当てられる。

もちろん、純粋なラムダ計算はチューリング完全であるため、これらのプリミティブのための**正確な**型解析は無理である。例えば、次のようなプログラム

```
if <長く複雑な計算> then true else (λx.x)
```

がブール値と関数のどちらを返すのか正しく決定するには、長く複雑な計算を実際に実行して、true と false のどちらが返るのかを見るしかない。しかし一般に、この長く複雑な計算は発散することさえあり、したがって、その計算結果を正確に予測しようとするいかなる型検査器もまた発散する。

ブール値の型システムを拡張して関数を含むようにするには、関数に評価される項を分類する型が明らかに必要である。差し当たり、その型を → と呼ぼう。もし、すべての λ 抽象に型 → を割り当てる型付け規則

$$\lambda x.t : \to$$

を追加するならば、λx.x のような単純な項と if true then (λx.true) else (λx.λy.y) のような複合的な項の両方を、関数を返すものとして分類できる。

しかし、この粗い解析は明らかに保守的すぎる。λx.true や λx.λy.y のような関数が型 → へ一緒くたにされてしまい、前者を true に適用するとブール値が返される一方、後者を true に適用すると関数が返されるという事実が無視される。一般に、有用な型を関数適用に割り当てるには、単に適用の左側が関数であるという以上のこと、つまり、その関数がどんな型を返すのかを知る必要

†1 （訳注）矢印型（arrow type）とも呼ばれる。

がある。さらに、関数が呼び出されたときに正しく振る舞うことを確信するには、引数にどの型を期待しているか追跡する必要がある。この情報を追跡するために、むきだしの型 → を、$T_1 \to T_2$ の形をした無限の型の族に置き換える。各 $T_1 \to T_2$ には、型 T_1 の引数を要求し、型 T_2 の結果を返す関数が分類される。

定義 9.1.1. 型 Bool 上の**単純型**の集合は次の文法により生成される。

T ::=		型：
	T→T	関数の型
	Bool	ブール値型

型構築子 → は右結合的である。すなわち、式 $T_1 \to T_2 \to T_3$ は $T_1 \to (T_2 \to T_3)$ を表す。 □

例えば Bool→Bool はブール値の引数をブール値の結果に写す関数の型である。(Bool→Bool)→(Bool→Bool)（または、(Bool→Bool)→Bool→Bool でも同じ）はブール値からブール値への関数を引数に取り、同じ型の結果を返す関数の型である。

9.2 型付け関係

λx.t のようなラムダ抽象に型を割り当てるには、そのラムダ抽象が引数に適用されると何が起こるのかを求める必要がある。その次に生まれる疑問は、引数に期待されている型をどのように知るかである。二つの回答がありうる。一つは λ 抽象が引数に意図している型を単に注釈してしまう方法、もう一つはラムダ抽象の本体を解析して引数の使われ方を解析し、そこから引数の取るべき型を演繹してみる方法である。差し当たり、前者を採用する。単なる λx.t の代わりに、$\lambda x{:}T_1.t_2$ と書く。この束縛変数の注釈から、引数が型 T_1 を持つと仮定していることがわかる。

一般に、項の型注釈により型検査器を誘導する言語は**明示的に型付けされた**言語と呼ばれる。型検査器にこの情報を**推論**、つまり**再構築**させる言語は**暗黙に型付けされている**といわれる（λ 計算の文献では、**型割り当て体系**という用語が使われることもある）。本書では主に明示的に型付けされた言語を中心的に扱う。暗黙の型付けは第 22 章で取り上げる。

λ 抽象の引数の型がわかれば、関数の結果の型が単に本体 t_2 の型になることは明らかである。ただし、t_2 における x の出現は型 T_1 の項を表すと仮定する。この直観は次の型付け規則で表現される。

$$\frac{x{:}T_1 \vdash t_2 : T_2}{\vdash \lambda x{:}T_1.t_2 : T_1 \to T_2} \qquad \text{(T-ABS)}$$

項はさらに λ 抽象を入れ子で持ちうるため、一般にはこのような仮定を複数考える必要がある。このため型付け関係は二項関係 t : T から三項関係 $\Gamma \vdash t : T$ に変更される。ここで Γ は t に現れる自由変数に関する仮定の集合である。

形式的には、**型付け文脈**（**型環境**とも呼ぶ）Γ は変数とその型の列であり、「コンマ」演算子は Γ の右に新しい束縛を加えて拡張する。空の文脈は ∅ と書かれることもあるが、通常は省略して $\vdash t : T$ と書き「閉じた項 t は空集合の仮定の下で型 T を持つ」ことを表す。

新しい束縛と既に Γ に現れている束縛を混同しないよう、名前 x は Γ で束縛されている変数とは異なるように選ばれるものとする。慣習により λ 抽象で束縛される変数はいつでも便宜的に名前を付け替えてよいため、この条件は必要に応じて束縛変数の名前を付け替えることにより常に満たせる。したがって、Γ は変数から型への有限な関数と考えてもよい。この直観に基づき、Γ で束縛される変数の集合を $dom(\Gamma)$ と書く。

ラムダ抽象に型を付ける規則の一般的な形は

$$\frac{\Gamma, \mathtt{x}{:}\mathtt{T}_1 \vdash \mathtt{t}_2 : \mathtt{T}_2}{\Gamma \vdash \lambda \mathtt{x}{:}\mathtt{T}_1.\mathtt{t}_2 : \mathtt{T}_1 \to \mathtt{T}_2} \qquad \text{(T-ABS)}$$

であり、前提は、結論の仮定にもう一つの仮定を加えている。

この議論から直ちに、変数の型付け規則が決まる。変数はその場で仮定されている型を持つ、という規則である。

$$\frac{\mathtt{x}{:}\mathtt{T} \in \Gamma}{\Gamma \vdash \mathtt{x} : \mathtt{T}} \qquad \text{(T-VAR)}$$

前提 $\mathtt{x}{:}\mathtt{T} \in \Gamma$ は「Γ において x に仮定された型は T である」と読む。

最後に、関数適用の型付け規則が必要である。

$$\frac{\Gamma \vdash \mathtt{t}_1 : \mathtt{T}_{11} \to \mathtt{T}_{12} \qquad \Gamma \vdash \mathtt{t}_2 : \mathtt{T}_{11}}{\Gamma \vdash \mathtt{t}_1\ \mathtt{t}_2 : \mathtt{T}_{12}} \qquad \text{(T-APP)}$$

もし $\mathtt{t}_1$ が（自由変数で表される値が Γ で仮定する型を持つという仮定の下で）$\mathtt{T}_{11}$ の引数を $\mathtt{T}_{12}$ の計算結果に写す関数へ評価され、かつ、もし $\mathtt{t}_2$ が型 $\mathtt{T}_{11}$ の計算結果に評価されるならば、$\mathtt{t}_1$ を $\mathtt{t}_2$ に適用した計算結果は型 $\mathtt{T}_{12}$ の値である。

ブール定数と条件式の型付け規則は以前（図 8-1）と同じである。もっとも、条件式の規則

$$\frac{\Gamma \vdash \mathtt{t}_1 : \mathtt{Bool} \qquad \Gamma \vdash \mathtt{t}_2 : \mathtt{T} \qquad \Gamma \vdash \mathtt{t}_3 : \mathtt{T}}{\Gamma \vdash \mathtt{if}\ \mathtt{t}_1\ \mathtt{then}\ \mathtt{t}_2\ \mathtt{else}\ \mathtt{t}_3 : \mathtt{T}} \qquad \text{(T-IF)}$$

に現れるメタ変数 T は、今や任意の関数型に具体化されうるため、分岐の結果が関数である条件式も型付けできることに注意されたい[†2]。

```
  if true then (λx:Bool. x) else (λx:Bool. not x);
▶ (λx:Bool. x) : Bool → Bool
```

これらの型付け規則を図 9-1 に要約する（完全を期すために構文と評価規則も含めた）。図において網掛けされた領域は、型無しラムダ計算に対して新しい構成要素を表す。これには、新しい規則と、古い規則に新しく追加された部分の両方がある。ブール値と数のときのように、計算全体を二つに分離した。つまり、この図に示す、基本型を全く持たない**純粋**単純型付きラムダ計算と、図 8-1 で既に示したブール値のための別の規則の集合である（もちろん図中のすべての型付け判断式に文脈 Γ を加えなければならない）。

多くの場合、記号 $\lambda_\to$ で単純型付きラムダ計算を表す（異なる基本型の集合を持つ体系でも同じ記号を用いる）。

演習 9.2.1. [★] 基本型を持たない純粋な単純型付きラムダ計算は、実際のところ、正しく型付けされた項が一つもないという意味で**縮退**している。これはなぜか。 □

$\lambda_\to$ の型付け規則のインスタンスは、型付き算術式に対して行ったのと同様に、組み合わせて**導出木**にすることができる。例えば、次は項 `(λx:Bool.x) true` が空の文脈で型 `Bool` を持つことを示す導出である。

[†2] 今後、実装のサンプル実行を示す例では計算結果と型の両方を表示する（明らかである場合は、省略することもある）。

→（型付き）　　　　λ（図 5-3）に基づく

構文

$$
\begin{array}{llr}
\mathtt{t} & ::= & \text{項：}\\
 & \mathtt{x} & \text{変数}\\
 & \lambda\mathtt{x}:\mathtt{T}.\mathtt{t} & \text{ラムダ抽象}\\
 & \mathtt{t}\ \mathtt{t} & \text{関数適用}\\[1ex]
\mathtt{v} & ::= & \text{値：}\\
 & \lambda\mathtt{x}:\mathtt{T}.\mathtt{t} & \text{ラムダ抽象値}\\[1ex]
\mathtt{T} & ::= & \text{型：}\\
 & \mathtt{T}\to\mathtt{T} & \text{関数の型}\\[1ex]
\Gamma & ::= & \text{文脈：}\\
 & \varnothing & \text{空の文脈}\\
 & \Gamma,\mathtt{x}:\mathtt{T} & \text{項変数の束縛}
\end{array}
$$

評価　　$\boxed{\mathtt{t} \longrightarrow \mathtt{t}'}$

$$\frac{\mathtt{t}_1 \longrightarrow \mathtt{t}_1'}{\mathtt{t}_1\ \mathtt{t}_2 \longrightarrow \mathtt{t}_1'\ \mathtt{t}_2} \quad \text{(E-APP1)}$$

$$\frac{\mathtt{t}_2 \longrightarrow \mathtt{t}_2'}{\mathtt{v}_1\ \mathtt{t}_2 \longrightarrow \mathtt{v}_1\ \mathtt{t}_2'} \quad \text{(E-APP2)}$$

$$(\lambda\mathtt{x}:\mathtt{T}_{11}.\mathtt{t}_{12})\ \mathtt{v}_2 \longrightarrow [\mathtt{x}\mapsto\mathtt{v}_2]\mathtt{t}_{12} \quad \text{(E-APPABS)}$$

型付け　　$\boxed{\Gamma \vdash \mathtt{t} : \mathtt{T}}$

$$\frac{\mathtt{x}:\mathtt{T} \in \Gamma}{\Gamma \vdash \mathtt{x} : \mathtt{T}} \quad \text{(T-VAR)}$$

$$\frac{\Gamma, \mathtt{x}:\mathtt{T}_1 \vdash \mathtt{t}_2 : \mathtt{T}_2}{\Gamma \vdash \lambda\mathtt{x}:\mathtt{T}_1.\mathtt{t}_2 : \mathtt{T}_1\to\mathtt{T}_2} \quad \text{(T-ABS)}$$

$$\frac{\Gamma \vdash \mathtt{t}_1 : \mathtt{T}_{11}\to\mathtt{T}_{12} \qquad \Gamma \vdash \mathtt{t}_2 : \mathtt{T}_{11}}{\Gamma \vdash \mathtt{t}_1\ \mathtt{t}_2 : \mathtt{T}_{12}} \quad \text{(T-APP)}$$

▲ 図 9-1.　純粋単純型付きラムダ計算（$\lambda_\to$）

$$
\dfrac{\dfrac{\dfrac{\mathtt{x}:\mathtt{Bool} \in \mathtt{x}:\mathtt{Bool}}{\mathtt{x}:\mathtt{Bool} \vdash \mathtt{x} : \mathtt{Bool}}\ \text{T-VAR}}{\vdash \lambda\mathtt{x}:\mathtt{Bool}.\mathtt{x} : \mathtt{Bool}\to\mathtt{Bool}}\ \text{T-ABS} \qquad \dfrac{}{\vdash \mathtt{true} : \mathtt{Bool}}\ \text{T-TRUE}}{\vdash (\lambda\mathtt{x}:\mathtt{Bool}.\mathtt{x})\ \mathtt{true} : \mathtt{Bool}}\ \text{T-APP}
$$

演習 9.2.2. [★ ↛]　次の項が与えられた型を持つことを、導出木を書くことにより示せ。

(1)　$\mathtt{f}:\mathtt{Bool}\to\mathtt{Bool} \vdash \mathtt{f}\ (\mathtt{if\ false\ then\ true\ else\ false}) : \mathtt{Bool}$

(2)　$\mathtt{f}:\mathtt{Bool}\to\mathtt{Bool} \vdash \lambda\mathtt{x}:\mathtt{Bool}.\ \mathtt{f}\ (\mathtt{if\ x\ then\ false\ else\ x}) : \mathtt{Bool}\to\mathtt{Bool}$　□

演習 9.2.3. [★]　f x y が Bool となるような文脈 Γ を見つけよ。そのような**すべての**文脈の集合を表す単純な記述を与えてみよ。　□

9.3 型付けの性質

第 8 章と同様に、型安全性を証明する前に、幾つかの基本的な補題を示す必要がある。ほとんどは以前と同様であり、型付け関係に文脈を加え、各証明に λ 抽象、関数適用と変数の節を加えるだけでよい。唯一の、新しくかつ重要な要求は、型付け関係のための**代入補題**である（補題 9.3.8）。

最初に、**逆転補題**は型付け導出が構築される方法についての観察を述べる。それぞれの節は構文形式ごとに、「もしこの形の項が正しく型付けされるならば、部分項はこの形の型を持つはずである……」といったことを示している。

補題 9.3.1.［型付け関係の逆転］

(1) $\Gamma \vdash$ x : R ならば x:R $\in \Gamma$ である。
(2) $\Gamma \vdash \lambda$x:T_1. t_2 : R ならば、ある Γ, x:$T_1 \vdash t_2$: R_2 を満たす R_2 が存在し R = $T_1 \to R_2$ である。
(3) $\Gamma \vdash t_1\ t_2$: R ならば、$\Gamma \vdash t_1$: $T_{11} \to$R と $\Gamma \vdash t_2$: T_{11} を満たす T_{11} が存在する。
(4) $\Gamma \vdash$ true : R ならば R = Bool である。
(5) $\Gamma \vdash$ false : R ならば R = Bool である。
(6) $\Gamma \vdash$ if t_1 then t_2 else t_3 : R ならば $\Gamma \vdash t_1$: Bool かつ $\Gamma \vdash t_2, t_3$: R である。 □

［証明］ 型付け関係の定義より明らか。 □

演習 9.3.2.［推奨, ★★★］ $\Gamma \vdash$ x x : T となるような文脈 Γ と型 T は存在するか。もし存在するならば Γ と T を与え、$\Gamma \vdash$ x x : T の導出を示せ。存在しないならば、存在しないことを証明せよ。

□

9.2 節では、型検査の仕事を単純にするために明示的に型付けされた計算体系の表現を選んだ。これは、関数抽象の束縛変数への型注釈の追加を伴ったが、他の場所には不要だった。これはどのような意味において「十分」なのだろうか？ 一つの答えは「型の一意性」定理により与えられる。この定理は、ある正しく型付けされた項とその型付け導出に一対一対応が付くことを教えてくれる。すなわち、型付け導出は項から一意に復元できる（そして、もちろん逆も成り立つ）。事実、この対応はとても単純で、ある意味では項と導出にはほとんど違いがない。

定理 9.3.3.［型の一意性］ 型付け文脈 Γ が一つ与えられると、それに対して一つの項 t は高々一つの型を持つ（ただし t の自由変数はすべて Γ の定義域に含まれるとする）。すなわち、ある項が型付け可能ならば、その型は一意に定まる。さらに、型付け関係を生成する推論規則からただ一つの型付け導出が構築できる。 □

［証明］ 演習［推奨, ★★★］。実のところ証明は何も言わなくてよいくらい直接的である。しかし詳細をある程度書き出すことは、型付け関係に関する証明の良い「準備運動」になるだろう。 □

以降で扱う多くの型システムでは、このような項と導出の単純な対応は成り立たない。ある一つの項に多くの型が割り当てられ、それぞれが多くの型付け導出で正当化されるかもしれないのである。これらの型システムでは、項から型付け導出を実効的に復元できることを示すため、しばしばかなりの労力を要する。

次の標準形補題は、さまざまな型について、その型の値が取りうる形を示してくれる。

補題 9.3.4.［標準形］

(1) v が型 Bool の値ならば、v は true か false のどちらかである。
(2) v が型 $T_1 \to T_2$ の値ならば、v = λx:T_1.t_2 である。 □

［証明］ ごく簡単である。（算術式の標準形補題（補題 8.3.1）の証明と同様。） □

標準形補題を用いて、定理 8.3.2 に似た進行定理を証明できる。定理の言明には少し変更が必要である。これは、自由変数を持たない**閉じた**項にのみ関心があるためである。開いた項については、進行定理は成立しない。例えば、f true は正規形であるが、値ではない。しかしながら、この失敗は言語の欠陥を意味しない。なぜなら完全なプログラム、つまり評価に関心がある項は常に閉じているためである。

定理 9.3.5.［進行］ $\mathtt{t}$ を閉じた、正しく型付けされた項とする（すなわち、ある T について $\vdash \mathtt{t} : \mathsf{T}$）。このとき $\mathtt{t}$ は値であるか、$\mathtt{t} \longrightarrow \mathtt{t}'$ なる $\mathtt{t}'$ が存在する。 □

［証明］ 型付け導出に関するごく簡単な帰納法による。ブール定数と条件式は型付き算術式における証明（定理 8.3.2）と全く同じである。変数は出現しない（$\mathtt{t}$ は閉じているため）。ラムダ抽象は値であるため、その場合は直ちに成り立つ。

唯一興味深いのは関数適用の場合である。$\mathtt{t} = \mathtt{t}_1\ \mathtt{t}_2$ で $\vdash \mathtt{t}_1 : \mathsf{T}_{11} \to \mathsf{T}_{12}$ かつ $\vdash \mathtt{t}_2 : \mathsf{T}_{11}$ とする。帰納法の仮定により、$\mathtt{t}_1$ は値か、さもなくば 1 ステップ評価が可能である。$\mathtt{t}_2$ についても同様である。もし $\mathtt{t}_1$ が 1 ステップ進むならば、規則 E-App1 が $\mathtt{t}$ に適用される。もし $\mathtt{t}_1$ が値で $\mathtt{t}_2$ が 1 ステップ進むならば、規則 E-App2 が適用される。最後に、もし $\mathtt{t}_1$ と $\mathtt{t}_2$ の両方が値ならば、標準形補題により $\mathtt{t}_1$ は $\lambda \mathtt{x}{:}\mathsf{T}_{11}.\mathtt{t}_{12}$ の形であるとわかり、規則 E-AppAbs が $\mathtt{t}$ に適用される。 □

次の仕事は、評価が型を保存することの証明である。まずはじめに、幾つかの「構造的補題」について述べる。これらの補題自体は特に面白くはないが、以降の証明において、型付け導出に対して便利な操作ができるようになる。

一つ目の構造的補題は、文脈の要素を都合のよいように並べ替えても、その文脈の下で導出できる型付け判断式の集合は変わらないということである。文脈におけるすべての束縛は別々の名前を持つこと（76 ページ）、また、文脈に束縛を加えるときは常に、束縛された名前が、既に束縛されている他の名前とは異なることを暗黙に仮定していること（慣習 5.3.4 を用いて必要に応じ名前を付け替える）を思い出そう。

補題 9.3.6.［並べ替え］ $\Gamma \vdash \mathtt{t} : \mathsf{T}$ かつ Δ が Γ の並べ替えならば、$\Delta \vdash \mathtt{t} : \mathsf{T}$ である。さらに、後者の導出は前者の導出と同じ深さ[†3]を持つ。 □

［証明］ 型付け導出に関するごく簡単な帰納法による。 □

補題 9.3.7.［弱化］ $\Gamma \vdash \mathtt{t} : \mathsf{T}$ かつ $\mathtt{x} \notin dom(\Gamma)$ ならば $\Gamma, \mathtt{x}{:}\mathsf{S} \vdash \mathtt{t} : \mathsf{T}$ である。さらに、後者の導出は前者の導出と同じ深さを持つ。 □

［証明］ 型付け導出に関するごく簡単な帰納法による。 □

これらの技術的な補題を用い、型付け関係に必要不可欠な性質を証明できる。その性質とは、適切な型を持つ項が変数に代入されたとき、正しい型付けが保存されるという性質である。同様の補題はプログラミング言語の安全性の証明において広範な役割を果たし、しばしば単に「代入補題」と呼ばれる。

補題 9.3.8.［代入の下での型の保存］ $\Gamma, \mathtt{x}{:}\mathsf{S} \vdash \mathtt{t} : \mathsf{T}$ かつ $\Gamma \vdash \mathtt{s} : \mathsf{S}$ ならば $\Gamma \vdash [\mathtt{x} \mapsto \mathtt{s}]\mathtt{t} : \mathsf{T}$ である。 □

［証明］ 型付け判断式 $\Gamma, \mathtt{x}{:}\mathsf{S} \vdash \mathtt{t} : \mathsf{T}$ の導出の深さに関する帰納法による。導出の証明において、最後に使われた型付け規則の場合分けによって進める[†4]。最も興味深いのは変数とラムダ抽象の場合である。

---

†3（訳注）導出木の高さのこと。

†4 もしくは、$\mathtt{t}$ が取りうる形に関する場合分けでもよい。これは構文の構築子ごとにちょうど一つの型付け規則があるためである。

T-VAR の場合 $\mathtt{t} = \mathtt{z}$
ただし $\mathtt{z}:\mathtt{T} \in (\Gamma, \mathtt{x}:\mathtt{S})$

考慮すべき場合が二つある。$\mathtt{z}$ が $\mathtt{x}$ である場合と、他の変数である場合である。もし $\mathtt{z} = \mathtt{x}$ ならば、$[\mathtt{x} \mapsto \mathtt{s}]\mathtt{z} = \mathtt{s}$。所要の結論は $\Gamma \vdash \mathtt{s} : \mathtt{S}$ であるが、それはこの補題の仮定に含まれている。もう一つの場合では、$[\mathtt{x} \mapsto \mathtt{s}]\mathtt{z} = \mathtt{z}$ であり、直ちに所望の結果が得られる。

T-ABS の場合 $\mathtt{t} = \lambda \mathtt{y}:\mathtt{T}_2.\mathtt{t}_1$
$\mathtt{T} = \mathtt{T}_2 \to \mathtt{T}_1$
$\Gamma, \mathtt{x}:\mathtt{S}, \mathtt{y}:\mathtt{T}_2 \vdash \mathtt{t}_1 : \mathtt{T}_1$

慣習 5.3.4 により、$\mathtt{x} \neq \mathtt{y}$ かつ $\mathtt{y} \notin FV(\mathtt{s})$ を仮定してよい。与えられた部分導出に並べ替えを用いると、$\Gamma, \mathtt{y}:\mathtt{T}_2, \mathtt{x}:\mathtt{S} \vdash \mathtt{t}_1 : \mathtt{T}_1$ が得られる。もう一方の導出（$\Gamma \vdash \mathtt{s} : \mathtt{S}$）に弱化を用いれば、$\Gamma, \mathtt{y}:\mathtt{T}_2 \vdash \mathtt{s} : \mathtt{S}$ が得られる。ここで帰納法の仮定より、$\Gamma, \mathtt{y}:\mathtt{T}_2 \vdash [\mathtt{x} \mapsto \mathtt{s}]\mathtt{t}_1 : \mathtt{T}_1$。T-ABS より、$\Gamma \vdash \lambda \mathtt{y}:\mathtt{T}_2.\ [\mathtt{x} \mapsto \mathtt{s}]\mathtt{t}_1 : \mathtt{T}_2 \to \mathtt{T}_1$。まさにこれが所要の結果である。なぜなら、代入の定義より、$[\mathtt{x} \mapsto \mathtt{s}]\mathtt{t} = \lambda \mathtt{y}:\mathtt{T}_2.\ [\mathtt{x} \mapsto \mathtt{s}]\mathtt{t}_1$ だからである。

T-APP の場合 $\mathtt{t} = \mathtt{t}_1\ \mathtt{t}_2$
$\Gamma, \mathtt{x}:\mathtt{S} \vdash \mathtt{t}_1 : \mathtt{T}_2 \to \mathtt{T}_1$
$\Gamma, \mathtt{x}:\mathtt{S} \vdash \mathtt{t}_2 : \mathtt{T}_2$
$\mathtt{T} = \mathtt{T}_1$

帰納法の仮定より、$\Gamma \vdash [\mathtt{x} \mapsto \mathtt{s}]\mathtt{t}_1 : \mathtt{T}_2 \to \mathtt{T}_1$ かつ $\Gamma \vdash [\mathtt{x} \mapsto \mathtt{s}]\mathtt{t}_2 : \mathtt{T}_2$。T-APP より、$\Gamma \vdash [\mathtt{x} \mapsto \mathtt{s}]\mathtt{t}_1\ [\mathtt{x} \mapsto \mathtt{s}]\mathtt{t}_2 : \mathtt{T}$、すなわち、$\Gamma \vdash [\mathtt{x} \mapsto \mathtt{s}](\mathtt{t}_1\ \mathtt{t}_2) : \mathtt{T}$ である。

T-TRUE の場合 $\mathtt{t} = \mathtt{true}$
$\mathtt{T} = \mathtt{Bool}$

このとき $[\mathtt{x} \mapsto \mathtt{s}]\mathtt{t} = \mathtt{true}$ であり、所望の結果 $\Gamma \vdash [\mathtt{x} \mapsto \mathtt{s}]\mathtt{t} : \mathtt{T}$ は直ちに成り立つ。

T-FALSE の場合 $\mathtt{t} = \mathtt{false}$
$\mathtt{T} = \mathtt{Bool}$

同様。

T-IF の場合 $\mathtt{t} = \mathtt{if}\ \mathtt{t}_1\ \mathtt{then}\ \mathtt{t}_2\ \mathtt{else}\ \mathtt{t}_3$
$\Gamma, \mathtt{x}:\mathtt{S} \vdash \mathtt{t}_1 : \mathtt{Bool}$
$\Gamma, \mathtt{x}:\mathtt{S} \vdash \mathtt{t}_2 : \mathtt{T}$
$\Gamma, \mathtt{x}:\mathtt{S} \vdash \mathtt{t}_3 : \mathtt{T}$

帰納法の仮定を三回用いれば次が得られる。

$\Gamma \vdash [\mathtt{x} \mapsto \mathtt{s}]\mathtt{t}_1 : \mathtt{Bool}$
$\Gamma \vdash [\mathtt{x} \mapsto \mathtt{s}]\mathtt{t}_2 : \mathtt{T}$
$\Gamma \vdash [\mathtt{x} \mapsto \mathtt{s}]\mathtt{t}_3 : \mathtt{T}$

これらから、T-IF より結果が得られる。 □

代入補題を使えば、型安全性の残り半分、つまり評価が正しい型付けを保存することを証明できる。

定理 9.3.9.［型保存］ $\Gamma \vdash \mathtt{t} : \mathtt{T}$ かつ $\mathtt{t} \longrightarrow \mathtt{t}'$ ならば $\Gamma \vdash \mathtt{t}' : \mathtt{T}$[†5] □

［証明］ 演習［推奨, ★★★］。証明の構造は、代入補題の利用を除けば、算術式の型保存の証明（定理 8.3.3）ととてもよく似ている。 □

[†5]（訳注：原著正誤表より。本書の他の型保存定理も同様）この定理に間違いはないが、本書の用途であればこれほど強力な定理は必要ない。(a) 閉じた項の評価のみを扱い、(b) 評価規則は値呼びである（このため、閉じた項の評価では、その閉じた部分項しか評価されない）ため、非空な Γ まで対象にする必要はない。

演習 9.3.10. [推奨, ★★] 演習 8.3.6 では型付き算術式の計算体系についての**主部展開**について調べた。これは単純型付きラムダ計算の「関数的な部分」では成り立つだろうか。言い換えると、t が条件式を含まないと仮定したとき、t ⟶ t′ かつ Γ ⊢ t′ : T ならば Γ ⊢ t : T だろうか。 □

9.4 Curry-Howard 対応

型構築子 "→" は次の二種類の型付け規則を伴う。

(1) **導入規則**（T-ABS）は、その型の要素がどのように**作られるか**を記述し、
(2) **除去規則**（T-APP）は、その型の要素がどのように**使われるか**を記述する。

導入形式（λ）が除去形式（関数適用）の直接の部分項であるとき、その結果は計算の機会を表す簡約基となる。

導入形式と除去形式という用語は型システムを議論するときにしばしば有用である。後の章ではより複雑な体系を扱うが、各型構築子の導入規則と除去規則に同様の結びつきがあることがわかるだろう。

演習 9.4.1. [★] 図 8-1 において型 Bool のどれが導入規則でどれが除去規則だろうか。図 8-2 の Nat についてはどうか。 □

導入形式・除去形式の用語は型理論と論理のつながりに由来する。このつながりは **Curry-Howard 対応**あるいは **Curry-Howard 同型**として知られている（Curry and Feys [128]; Howard [224]）。これは簡潔にいうと、構成的論理においては命題 P の証明が P の具体的な**証拠**からなるという考え方である[†6]。Curry と Howard が気づいたのは、そのような証拠は計算のような雰囲気を強く持っているということだった。例えば、命題 $P \supset Q$ の証明は、P の証明から、Q の証明を構築する機械的な手続きとみなせる。または、望むなら、P の証明に関して**抽象化**された Q の証明ともみなせる。同様に、$P \wedge Q$ の証明は P の証明と Q の証明を組み合わせたものからなる。この観察から、次の対応関係が得られる。

論理	プログラミング言語
命題	型
命題 $P \supset Q$	型 P→Q
命題 $P \wedge Q$	型 P×Q（11.6 節を参照）
P の証明	型 P の項 t
命題 P が証明可能	型 P を持つ項が存在する

この見方において、単純型付きラムダ計算の項は、その型に対応する論理的命題の証明である。計算すなわちラムダ項の簡約は、**カット除去**による証明の単純化という論理的な操作に対応する。Curry-Howard 対応は「**命題＝型**」の類推とも呼ばれる。この対応についての深い議論は Girard, Lafont, and Taylor [179]; Gallier [166]; Sørensen and Urzyczyn [440]; Pfenning [362]; Goubault-Larrecq and Mackie [185]; Simmons [434] など多くの文献でなされている。

---

†6 古典論理と構成的論理の特徴的な違いは、後者は**排中律**のような証明規則を持たないことである。排中律とは、すべての命題 Q について、Q か $\neg Q$ のどちらか一方が成り立つことを指す。構成的論理の下で $Q \vee \neg Q$ を証明するには、Q か $\neg Q$ のどちらかの証拠を与えなければならない。

Curry-Howard 対応の美しさは、特定の型システムとそれに関連する論理体系にとどまらない。それどころか、それは膨大な種類の型システムと論理体系に拡張できる。例えば、System F（第 23 章）は、パラメータ多相により型の量化を伴うが、これは命題の量化を許す二階の構成的論理に正確に対応する。System F_ω（第 30 章）は、ある高階論理に対応する。実際、この対応は新しい成果を分野をまたいで移転するためにしばしば利用された。そのようにして、Girard の**線形論理** [178] から線形型システムの着想がもたらされた（Wadler [480]; Wadler [481]; Turner, Wadler, and Mossin [468]; Hodas [212]; Mackie [293]; Chirimar, Gunter, and Riecke [98]; Kobayashi, Pierce, and Turner [265]; 他多数）。一方、**様相論理**は**部分評価**と**実行時コード生成**の枠組みを考えるために用いられた（Davies and Pfenning [133]; Wickline, Lee, Pfenning, and Davies [494] およびこれらが参照している文献を参照）。

9.5 型消去と型付け可能性

図 9-1 では、評価関係を単純型付き項の上に直接定義したが、型注釈は評価において何の役割も果たさない。つまり、関数が適用される引数の型が適切であることを確認するような実行時検査は何ら行わない。しかしながら、項の評価は中に含まれる注釈を付けたままで行う。

本格的なプログラミング言語のコンパイラのほとんどは、実行時には注釈が伴わないようにしている。注釈は型検査時に用いられる（多くの洗練されたコンパイラはコード生成時にも用いる）が、コンパイル後のプログラムには現れない。事実上、プログラムは評価される前に型無しの形式に戻される。このスタイルの意味論は、単純型付き項を型無しの項へ写す**消去**関数によって形式化される。

定義 9.5.1. 単純型付き項 t の**型消去**は次のように定義される。

$$
\begin{aligned}
\mathit{erase}(\mathtt{x}) &= \mathtt{x}\\
\mathit{erase}(\lambda\mathtt{x}{:}\mathtt{T_1}.\ \mathtt{t_2}) &= \lambda\mathtt{x}.\ \mathit{erase}(\mathtt{t_2})\\
\mathit{erase}(\mathtt{t_1}\ \mathtt{t_2}) &= \mathit{erase}(\mathtt{t_1})\ \mathit{erase}(\mathtt{t_2})
\end{aligned}
$$
□

もちろん、単純型付きラムダ計算の意味を表現する二通りの方法は一致すると期待したい。型付けされた項を直接に評価してもよいし、型消去した後の型無しの項を評価してもよいはずである。この期待は次の定理で形式化され、「評価は型消去と可換である」というスローガンで要約される。ここでの可換とは、それらの操作がどの順でも実行できること、つまり、評価してから型消去しても、型消去してから評価しても、同じ項が得られるという意味である。

定理 9.5.2.

(1) 型付きの評価関係の下で $\mathtt{t} \longrightarrow \mathtt{t'}$ ならば、$\mathit{erase}(\mathtt{t}) \longrightarrow \mathit{erase}(\mathtt{t'})$

(2) 型無しの評価関係の下で $\mathit{erase}(\mathtt{t}) \longrightarrow \mathtt{m'}$ ならば、ある単純型付き項 t′ が存在し、$\mathtt{t} \longrightarrow \mathtt{t'}$ かつ $\mathit{erase}(\mathtt{t'}) = \mathtt{m'}$ □

[証明] 評価導出に関する単純な帰納法による。 □

ここでの「コンパイル」は単純なので、定理 9.5.2 は自明といってよいほど明らかである。しかしながら、より興味深い言語やコンパイラでは、とても重要な性質になる。つまりこの性質は、プログラマが書く言語で直接表現された「高水準の」意味論が、他方の、言語の実装で実際に用いられる低水準の評価戦略と一致することを示しているのである。

型消去関数に由来する別の興味深い疑問がある。ある型無しの項 m について、型消去により m に

なるような、単純型付き項 t を見つけられるだろうか？

定義 9.5.3. ある型無しラムダ計算の項 m について、単純型付き項 t、型 T、文脈 Γ が存在し *erase*(t) = m かつ Γ ⊢ t : T のとき、m は $\lambda_{\to}$ で**型付け可能**であるという。 □

第 22 章で、これと密接にかかわる話題である $\lambda_{\to}$ の**型再構築**を考えるときに、この点についてより詳細に取り扱う。

9.6 Curry スタイル 対 Church スタイル

本章では単純型付きラムダ計算の意味論を定式化する二つの異なるスタイルを見てきた。単純型付き計算の構文の上に直接定義された評価関係と、型無し計算へのコンパイル＋型無しの項の上の評価関係である。二つのスタイルの重要な共通点は、項 t が実際に正しく型付けされているかどうかにかかわらず、t の振る舞いについて考えるのに意味があることである。この形式の言語定義はしばしば **Curry スタイル**と呼ばれる。まず最初に項を定義し、それらがどのように振る舞うかを示す意味論を定義し、そして望まぬ振る舞いを持つ項を拒否する型システムを与える。意味論が型付けに優先するのである。

言語の定義を構成するいくぶん違った方法は、まず項を定義し、正しく型付けされた項を識別し、それらに対してのみ意味を与えることである。**Church スタイル**と呼ばれるこれらの体系では型付けが意味論に優先する。「型付けできない項の振る舞いは何だろうか？」と問うことはありえない。それどころか、厳密に言えば、Church スタイルの体系で実際に評価されるのは型付け**導出**であり、項ではない（この例として 15.6 節を参照）。

歴史的に、暗黙に型付けされたラムダ計算の形式化は Curry スタイルにより与えられることが多く、Church スタイルによる形式化は明示的に型付けされた体系のみで一般的だった。このことは用語の混乱を引き起こした。時として、「Church スタイル」は明示的に型付けされた**構文**を、「Curry スタイル」は暗黙に型付けされた構文を指すのである。

9.7 注記

単純型付きラムダ計算は Hindley and Seldin [210] により研究され、Hindley [209] でさらに詳細にわたって研究されている。

正しく型付けされたプログラムは「おかしくなる（*wrong* になる）」ことはない。
—*Robin Milner, 1978* [307]

単純型の ML 実装

本章で解説する実装は、ブール値（図 8-1）を伴った単純型付きラムダ計算（図 9-1）に対応する。
この章のコードは、ウェブリポジトリでは simplebool 実装にある。

第 7 章で示した型無しラムダ計算の実装を踏襲し、$\lambda_{\rightarrow}$ を ML プログラムとして具体的に実現する。主に追加するものは、与えられた項の与えられた文脈における型を計算するための関数 typeof である。しかしそこに辿り着く前に、文脈を扱うための低水準な機構が少し必要である。

10.1 文脈

第 7 章（64 ページ）を思い出すと、文脈とは単に、変数名と束縛との二つ組のリストである。

```
type context = (string * binding) list
```

第 7 章では、構文解析と表示の際、項の名前付き形式と名前無し形式とを変換するためだけに、文脈を用いた。このため、変数の名前だけわかれば十分だった。binding 型は全く情報を持たない、構築子をただ一つ持つ自明なデータ型で、以下のように定義されていた。

```
type binding = NameBind
```

型検査器を実装するためには、文脈を用いて変数の型付けの仮定を保持する必要がある。そのため、VarBind と呼ばれる新しい構築子を binding 型に付け加える。

```
type binding =
    NameBind
  | VarBind of ty
```

各 VarBind 構築子は、対応する変数の型付けの仮定を保持する。VarBind に加えて元の NameBind 構築子を残しているのは、型付けの仮定に関与しない表示機能と構文解析機能のためである。（異なる実装戦略としては、二つの完全に異なった context 型を定義する方法もある。一つは構文解析と表示のため、もう一つは型検査のためである。）

typeof 関数は、関数 addbinding を用いて、新しい変数束縛 (x, bind) により文脈 ctx を拡張する。ただし、文脈はリストとして表現されているので、addbinding は本質的にはただの cons である。すなわち、

```
let addbinding ctx x bind = (x,bind)::ctx
```

である。逆に、文脈 ctx において特定の変数 i に対応する型の仮定を取り出すためには、以下の関数 getTypeFromContext を使う（ファイル情報 fi は、i が範囲外のときにエラーメッセージを表示するために用いられる）。

```
let getTypeFromContext fi ctx i =
  match getbinding fi ctx i with
      VarBind(tyT) → tyT
    | _ → error fi
      ("getTypeFromContext: Wrong kind of binding for variable "
        ^ (index2name fi ctx i))
```

match はある種の内部整合性を検査することとなる。普通の状況では、getTypeFromContext は、i 番目の束縛が実際に VarBind となっているような文脈に対して呼び出されるはずである。後の章では、しかし、他の形の束縛（特に、**型変数**に対する束縛）を追加するので、getTypeFromContext が誤った種類の変数に対して呼び出されることが起こりうる。その場合、低水準の error 関数を用いてメッセージを表示するが、エラーの起こったファイル位置を報告できるように info を渡す。

```
val error : info → string → 'a
```

関数 error の結果の型は、型変数 'a であって、任意の ML 型で具体化できる（メッセージを表示してプログラムを終了し、決して値を返さないのだから、これは理にかなっている）。ここでは、error の結果は、match のもう一方の分岐が返す型と同じ、ty であると仮定する必要がある。

項は内部的に名前無し形式で表現されており、変数は数値インデックスとして表現されているので、型付けの仮定を検索するときは名前ではなく**インデックス**を用いることに注意されたい。getbinding 関数は単に、与えられた文脈の中から i 番目の束縛を返せばよい。

```
val getbinding : info → context → int → binding
```

この関数の定義はこの本のウェブサイト上の simplebool 実装に含まれている。

10.2 項と型

型の構文は、図 8-1 と図 9-1 に示された抽象構文から ML データ型に直接翻訳できる。

```
type ty =
    TyArr of ty * ty
  | TyBool
```

項の表現は、TmAbs 節に型注釈を追加することを除けば、型無しラムダ計算（64 ページ）において用いたものと同じである。

```
type term =
    TmVar of info * int * int
  | TmAbs of info * string * ty * term
  | TmApp of info * term * term
  | TmTrue of info
  | TmFalse of info
  | TmIf of info * term * term * term
```

10.3 型検査

型検査関数 typeof は $\lambda_{\rightarrow}$（図 8-1 と図 9-1）に対する型付け規則の直訳とみなせる。もしくは、より正確には、逆転補題（補題 9.3.1）の写しとみなすこともできる。なぜなら、逆転補題は、各構文に対して、その形の項が正しく型付けされているためにはどのような条件が満たされていなければならないかを正確に教えてくれるからである。それらの型付け規則は、ある形の項がある条件の下で正しく型付けされるということは教えてくれるが、個々の型付け規則を見ても、ある項が正しく型付けされ**ない**と結論付けることは決してできない。なぜなら他の規則を用いればその項を型付けできる可能性が常にあるからである。（今のところ、これは些細な違いのように見えるかもしれない。逆転補題が型付け規則からとても直接的に導かれるためである。しかしこの違いは、逆転補題の証明が $\lambda_{\rightarrow}$ よりも難しい、後の体系において重要となる。）

```
let rec typeof ctx t =
  match t with
    TmVar(fi,i,_) → getTypeFromContext fi ctx i
  | TmAbs(fi,x,tyT1,t2) →
      let ctx' = addbinding ctx x (VarBind(tyT1)) in
      let tyT2 = typeof ctx' t2 in
      TyArr(tyT1, tyT2)
  | TmApp(fi,t1,t2) →
      let tyT1 = typeof ctx t1 in
      let tyT2 = typeof ctx t2 in
      (match tyT1 with
          TyArr(tyT11,tyT12) →
            if (=) tyT2 tyT11 then tyT12
            else error fi "parameter type mismatch"
        | _ → error fi "arrow type expected")
  | TmTrue(fi) →
      TyBool
  | TmFalse(fi) →
      TyBool
  | TmIf(fi,t1,t2,t3) →
     if (=) (typeof ctx t1) TyBool then
       let tyT2 = typeof ctx t2 in
       if (=) tyT2 (typeof ctx t3) then tyT2
       else error fi "arms of conditional have different types"
     else error fi "guard of conditional not a boolean"
```

OCaml 言語に関する、細かいことを幾つかここで触れておく。第一に、OCaml の等値演算子 = を括弧の中に書いたのは、通常の中置でなく、前置で用いるためである。後の typeof においては、型の比較操作を単なる等値性よりも洗練させる必要があり、それとの対比を容易にするためである。第二に、等値演算子は、複合的な値に対しては、**ポインタの**等値性ではなく**構造的な**等値性を計算する。つまり、以下の式

```
let t  = TmApp(t1,t2) in
let t' = TmApp(t1,t2) in
(=) t t'
```

は、TmApp の t と t' とに束縛された二つのインスタンスが、異なるタイミングで割り当てられ、メモリ上で異なるアドレスを持っているが、true を返すことが保証されている。

第 11 章

単純な拡張

本章で扱う体系は純粋型付きラムダ計算（図 9-1）のさまざまな拡張である。対応する OCaml 実装は `fullsimple` で、すべての拡張を含んでいる。

単純型付きラムダ計算は理論的性質を興味深くするには十分な構造を持つが、あまり大したプログラミング言語ではなかった。この章ではより馴染み深い言語とのギャップを埋めるために、馴染み深い機能を幾つか導入する。これらの機能は型付けのレベルでは容易に扱える。この章を通しての重要なテーマは**派生形式**の概念である。

11.1 基本型

どんなプログラミング言語にもさまざまな**基本型**、すなわち数やブール値や文字といった、構造を持たない単純な値の集合と、それらの値を操作するための適切なプリミティブ演算がある。本書では既に自然数やブール値について詳細に見てきた。他の基本型は、全く同じ方法で、言語設計者が望む数だけ追加できる。

本書では今後、例を味付けするスパイスとして、`Bool` と `Nat` の他に、`String`（要素は `"hello"` など）と `Float`（要素は `3.14159` など）を用いることがある。

理論的な目的のためには、個々の基本型とその演算の詳細は捨象してしまったほうがよい場合も多い。代わりに言語が単に**非解釈**あるいは**未知**の基本型の集合 $\mathcal{A}$ を備えており、それらの上にプリミティブ演算は全くないとするのである。そうするためには図 11-1 に示すように、単に $\mathcal{A}$ の要素（メタ変数 A で表す）を、型の集合に加えればよい。基本型の集合は $\mathcal{B}$ ではなく $\mathcal{A}$ という文字で表す。これは、与えられた体系にブール値が存在することを表すために使っている記号 $\mathbb{B}$ との混同を防ぐためである。$\mathcal{A}$ は**原子型**（atomic type）の略と考えてもよい。基本型は型システムに関する限り内部構造を持たないため、しばしば原子型とも呼ばれるのである。基本型の名前として A、B、C などを用いる。変数の場合と同様に、A は基本型と、基本型の上を動くメタ変数との両方に用い、個々の例でどちらを意図するかは文脈に依存するので注意されたい。

非解釈な型は役に立たないのだろうか？ そんなことは全くない。非解釈な型の要素を直に書くことはできないが、基本型の要素の上を動く変数を束縛することはできる。例えば、以下の関数[†1]

→ A　　　　　　　　　　　　　　　　λ→（図 9-1）の拡張

新しい構文形式	
T ::= ...	型：
A	基本型

▲ 図 11-1.　非解釈基本型

†1 以降では、評価結果を表示する際にスペースを節約するため、λ 抽象は省略して単に <fun> と書くことにする。

→ Unit　　　　$\lambda_{\to}$（図 9-1）の拡張

新しい構文形式

t	::=	...	項：
		unit	定数 unit
v	::=	...	値：
		unit	定数 unit
T	::=	...	型：
		Unit	Unit 型

新しい型付け規則　　$\Gamma \vdash$ t : T

$$\Gamma \vdash \texttt{unit} : \texttt{Unit} \quad \text{(T-UNIT)}$$

新しい派生形式

$$\texttt{t}_1\texttt{;t}_2 \overset{\text{def}}{=} (\lambda\texttt{x:Unit.t}_2)\ \texttt{t}_1 \quad \text{ただし } \texttt{x} \notin FV(\texttt{t}_2)$$

▲ 図 11-2.　Unit 型

```
  λx:A. x;
▸ <fun> : A → A
```

は、A の要素が何であれ、その上の恒等関数である。同様に、

```
  λx:B. x;
▸ <fun> : B → B
```

は、B 上の恒等関数である。一方

```
  λf:A→A. λx:A. f(f(x));
▸ <fun> : (A→A) → A → A
```

は、何らかの与えられた関数 f の振る舞いを、引数 x に対して二度繰り返すような関数である。

11.2　Unit 型

別の有用な基本型で、ML 系の言語に特に見られるものとして、図 11-2 に示す要素が一つしかない型 Unit がある。前節の非解釈な基本型と異なり、この型は可能な限り単純な方法で解釈される。すなわち、唯一の要素である項定数 unit（小文字の u で書く）と、unit を Unit の要素とする型付け規則を明示的に導入する。さらに、unit を、計算の結果としてありうる値の集合に加える。実際のところ、unit は Unit 型の式を評価した結果としてありうる**ただ一つの**値である。

Unit 型は、純粋関数型言語においても全く興味をひかないというわけではない[†2]が、主な応用は参照セルへの破壊的代入といった副作用（これについては第 13 章で述べる）を持つ言語にある。そのような言語では、式の副作用に関心があり、結果には関心がないことがある。Unit はそのような式の結果の型として適している。

このような Unit の使われ方は、C や Java での void 型の役割と似ている。void という名前からは空の型 Bot（15.4 節参照）を連想するが、void の使われ方は実際には我々の Unit に近い。

[†2] 以下のちょっとしたクイズは面白いかもしれない。

演習 11.2.1. [★★★]　基本型として Unit 型のみを持つ単純型付きラムダ計算で、以下の条件を満たす項の列 $\texttt{t}_1, \texttt{t}_2, \ldots$ を作ることはできるか。各 n に対して項 $\texttt{t}_n$ は高々 $O(n)$ のサイズを持つが、それを正規形まで評価するには少なくとも $O(2^n)$ ステップを必要とする。 □

11.3 派生形式：逐次実行とワイルドカード

副作用のある言語では、二つかそれ以上の式を順に評価できると便利なことが多い。**逐次実行記法** $\mathtt{t_1;t_2}$ は、$\mathtt{t_1}$ を評価し、その自明な結果を捨て、そして $\mathtt{t_2}$ の評価へと移るという効果を持つ。

逐次実行の形式化には実際には二通りの方法がある。一つは、これまで他の構文形式に対して用いてきたのと同じパターンに従うことである。すなわち、$\mathtt{t_1;t_2}$ を項の構文の定義における新たな選択肢として加え、意図する「;」の振る舞いを捉えた二つの評価規則

$$\frac{\mathtt{t_1} \longrightarrow \mathtt{t_1'}}{\mathtt{t_1;t_2} \longrightarrow \mathtt{t_1';t_2}} \qquad \text{(E-SEQ)}$$

$$\mathtt{v_1;t_2} \longrightarrow \mathtt{t_2} \qquad \text{(E-SEQNEXT)}$$

および型付け規則

$$\frac{\Gamma \vdash \mathtt{t_1} : \mathtt{Unit} \qquad \Gamma \vdash \mathtt{t_2} : \mathtt{T_2}}{\Gamma \vdash \mathtt{t_1;t_2} : \mathtt{T_2}} \qquad \text{(T-SEQ)}$$

を追加する。

逐次実行を形式化する別の方法は、$\mathtt{t_1;t_2}$ を単に $(\lambda\mathtt{x:Unit.t_2})\ \mathtt{t_1}$ という項（ここで変数 $\mathtt{x}$ は**フレッシュ**、すなわち $\mathtt{t_2}$ のすべての自由変数と異なる変数とする）の**略記**とみなす方法である。

どちらの表し方でも、プログラマにとって同じ結果になるのは、直観的にはかなり明らかである。すなわち、逐次実行のための高水準な型付け規則と評価規則は、$\mathtt{t_1;t_2}$ が $(\lambda\mathtt{x:Unit.t_2})\ \mathtt{t_1}$ の略記であることから**導出**できる。この直観的な対応をより形式的に捉えるには、型付けと評価の両方が、略記の展開操作と「可換」であると論じればよい。

定理 11.3.1.［逐次実行は派生形式である］ Unit 型、逐次実行、E-SEQ、E-SEQNEXT、T-SEQ の規則を持つ単純型付きラムダ計算を λ^E と書く（「E」は**外部言語**（external language）を表している）。また、Unit のみを持つ単純型付きラムダ計算を λ^I と書く（「I」は**内部言語**（internal language）を表す）。$e \in \lambda^E \to \lambda^I$ を、外部言語を内部言語に翻訳する**詳細化関数**とする。e は $\mathtt{t_1;t_2}$ のすべての出現を、$(\lambda\mathtt{x:Unit.t_2})\ \mathtt{t_1}$ へと置き換える（$\mathtt{x}$ はそれぞれの場合においてフレッシュに選ぶ）。すると、λ^E の各項 $\mathtt{t}$ に対して、

- $\mathtt{t} \longrightarrow_E \mathtt{t'}$ のとき $e(\mathtt{t}) \longrightarrow_I e(\mathtt{t'})$ であり、逆に $e(\mathtt{t}) \longrightarrow_I u$ であれば λ^E の項 $\mathtt{t'}$ が存在して $\mathtt{u} = e(\mathtt{t'})$ かつ $\mathtt{t} \longrightarrow_E \mathtt{t'}$
- $\Gamma \vdash^E \mathtt{t} : \mathtt{T}$ のとき、かつそのときに限り $\Gamma \vdash^I e(\mathtt{t}) : \mathtt{T}$

が成り立つ。ここで、λ^E と λ^I の評価規則と型付け規則は、どちらがどちらかわかるように、それぞれ E と I による注記を加えてある。 □

［証明］ いずれの命題の、いずれの向きの含意も、$\mathtt{t}$ の構造に関する単純な帰納法で証明できる。 □

定理 11.3.1 によって、**派生形式**という言葉を使うことが正当化される。なぜならば、この定理は、逐次実行の型付けと評価に関する振る舞いが、より基本的な操作であるラムダ抽象と関数適用から派生させられることを示しているためである。逐次実行のような機能を、独立した言語要素としてではなく派生形式として導入する利点は、表層構文（すなわちプログラマが実際にプログラムを書くのに用いる構文）を拡張しつつ、型安全性といった定理を証明しなくてはならない内部言語を複雑

→ as　　　　λ→（図 9-1）の拡張

新しい構文形式

t ::= ...　　項：

　t as T　　型指定

新しい評価規則　　$\boxed{t \longrightarrow t'}$

$$v_1 \text{ as } T \longrightarrow v_1 \qquad \text{(E-ASCRIBE)}$$

$$\frac{t_1 \longrightarrow t_1'}{t_1 \text{ as } T \longrightarrow t_1' \text{ as } T} \qquad \text{(E-ASCRIBE1)}$$

新しい型付け規則　　$\boxed{\Gamma \vdash t : T}$

$$\frac{\Gamma \vdash t_1 : T}{\Gamma \vdash t_1 \text{ as } T : T} \qquad \text{(T-ASCRIBE)}$$

▲ 図 11-3.　型指定

にせずに済むことである。言語の機能の記述を分解するこの手法は、Algol 60 report（Naur et al. [334]）に既に見られ、より最近の多くの言語定義、中でも、Definition of Standard ML（Milner, Tofte, and Harper [315]; Milner, Tofte, Harper, and MacQueen [316]）で多用されている。

派生形式はしばしば Landin に従って**糖衣構文**とも呼ばれる。派生形式をより低水準の定義によって置き換えることは**脱糖衣**と呼ばれる。

他の派生形式として、変数束縛子における「ワイルドカード」の慣習がある。これは後の例で便利である。例えば逐次実行を脱糖衣すると作られる項のように、実際には引数の変数を抽象の本体で使わないような、「ダミー」のラムダ抽象を書きたくなることは多い。こういう場合、束縛変数の名前を選ぶのは面倒であるため、**ワイルドカード束縛子**（_ と書く）を代わりに用いたい。すなわち、λ_:S.t を λx:S.t の略記として書くのである。ただし、ここで x は t に現れないような変数である。

演習 11.3.2. [★]　ワイルドカードを用いたラムダ抽象のための型付け規則と評価規則を定義し、それらが上記の略記から導出できることを証明せよ。 □

11.4　型指定

後でしばしば重宝する、別の単純な機能として、与えられた項に特定の型を明示的に指定 (ascribe) する機能が挙げられる。これはすなわち、プログラムテキスト中にその項がこの型であるというアサーションを記す機能である。型 T を指定した項 t を、「t as T」と書く。この構文のための型付け規則 T-ASCRIBE（図 11-3 参照）は、単に与えられた型 T が実際に t の型であることを確認するだけである。評価規則 E-ASCRIBE も同じくらい単純であり、単に型指定を取り除き、t を通常通り評価できるようにする。

プログラムを書く際に型指定が便利な場合はいろいろとある。一般的なものは**ドキュメント化**である。プログラムを読む際に大きな複合的な式の部分式の型を追うのは難しいことがあるが、型指定を賢く用いれば、そんなプログラムをずっと追いやすくできる。同様に、特に複雑な式の場合、**書き手**自身にとってすらすべての部分式の型が明らかでないときもある。型指定を少しばかりちりばめれば、プログラマの思考を明確化できる。実際、わかりにくい型エラーの源を突き止めるのに、型指定が役立つときがある。

型指定の別の用途として、複雑な型の**表示**の制御がある。この本の例を検査するために使っている型検査器（と、付属の OCaml による実装のうち、名前が `full` で始まるもの）は、長いか複雑な型式に対して略記を導入するための簡単な仕組みを提供している（他の実装では、読んだり変更したりを容易にするために、この仕組みは省いてある）。例えば、宣言

```
UU = Unit→Unit;
```

は、`Unit→Unit` の略記として `UU` を導入する。それ以降、`UU` を見たら常に、`Unit→Unit` とみなす。例えば、以下のように書ける。

```
(λf:UU. f unit) (λx:Unit. x);
```

型検査中は、これらの略記は必要に応じて自動的に展開される。逆に、型検査器は可能なときにはいつでも型を略記に折り畳もうとする（具体的には、部分項の型を計算するたびに、その型が現在定義されている略記のいずれかに正確に合致するか検査し、合致したらその型を略記に置き換える）。これは通常は妥当な結果を与えるが、時には違った風に型を表示したいときもある。単純な合致戦略だと型検査器が略記を折り畳む機会を損なってしまう場合（例えば、レコード型のフィールドの順序を並べ替え可能な体系の場合、この戦略は `{a:Bool,b:Nat}` が `{b:Nat,a:Bool}` と交換可能であることを認識しないだろう）や、あるいは、何か他の理由で違った風にその型を表示したい場合があるからである。例えば

```
λf:Unit→Unit. f;
▸ <fun> : (Unit→Unit) → UU
```

の場合、関数の結果の型は `UU` に折り畳まれるが、引数の型は `UU` へと折り畳まれない。これを `UU→UU` と表示したい場合には、ラムダ抽象の型注釈を以下のように変更すればよい。

```
λf:UU. f;
▸ <fun> : UU → UU
```

もしくは、ラムダ抽象全体に型指定を追加すればよい。

```
(λf:Unit→Unit. f) as UU→UU;
▸ <fun> : UU → UU
```

型検査器が型指定 `t as T` を処理する際、`t` が `T` 型を持つことを検査する間は `T` 中のすべての略記を展開するが、その後は書かれた通りの `T` をこの式の型として出力する。型指定を用いた型の表示の制御は、この本の実装の設計に特有の部分がいくぶんある。本格的なプログラミング言語では、略記と型の表示の仕組みは、不要である（例えば Java では、設計により、すべての型が短縮名で表される。第 19 章参照）か、もしくは言語自体に、より密に統合されている（例えば OCaml でのように。Rémy and Vouillon [406]; Vouillon [477] を参照）。

型指定の最後の使い方は、151 ページでより詳しく議論するが、**抽象化**の仕組みとしての使い方である。与えられた項 `t` が複数の異なる型を持ちうる体系（例えば部分型付けのある体系）では、型指定を使って、あたかも `t` がより少数の型だけを持つかのように扱うよう、型検査器に指示することで、それらの型の一部を「隠す」ことができる。型指定と**キャスト**の関係についても、15.5 節で述べる。

演習 11.4.1. [推奨, ★★]　(1) 型指定を派生形式として形式化せよ。「公式」の型付け規則および評価規則が、その形式化に何らかの適切な意味で対応することを証明せよ。(2) 評価規則 E-Ascribe と E-Ascribe1 の代わりに、以下の「先行的（eager）」な規則が与えられたとする。

$$\mathtt{t}_1 \ \mathtt{as}\ \mathtt{T} \longrightarrow \mathtt{t}_1 \qquad \text{(E-AscribeEager)}$$

→ let　　λ→（図 9-1）の拡張

新しい構文形式

t ::=	...	項：
	let x=t in t	let 束縛

新しい評価規則　　$\boxed{t \longrightarrow t'}$

$$\texttt{let } x\texttt{=}v_1 \texttt{ in } t_2 \longrightarrow [x \mapsto v_1]t_2 \quad \text{(E-LetV)}$$

$$\frac{t_1 \longrightarrow t_1'}{\texttt{let } x\texttt{=}t_1 \texttt{ in } t_2 \longrightarrow \texttt{let } x\texttt{=}t_1' \texttt{ in } t_2} \quad \text{(E-Let)}$$

新しい型付け規則　　$\boxed{\Gamma \vdash t : T}$

$$\frac{\Gamma \vdash t_1 : T_1 \qquad \Gamma, x{:}T_1 \vdash t_2 : T_2}{\Gamma \vdash \texttt{let } x\texttt{=}t_1 \texttt{ in } t_2 : T_2} \quad \text{(T-Let)}$$

▲ 図 11-4.　let 束縛

この規則は、型指定に到達し次第、型指定を除去する。この場合にも、型指定を派生形式として扱うことはできるか。□

11.5 let 束縛

複雑な式を書く際には、多くの場合、部分式に対して名前を付けると、繰り返しを避けたり、式を読みやすくしたりできる。ほとんどの言語はそのための手段を一つかそれ以上提供している。例えば、ML では $\texttt{let } x\texttt{=}t_1 \texttt{ in } t_2$ と書くと、「式 t_1 を評価し、その結果の値に名前 x を束縛した状態で t_2 を評価せよ」という意味になる。

我々の let 束縛子（図 11-4 に要約）は、ML と同様に値呼び評価順序を採用しており、let 束縛項は let の本体の評価前に完全に評価されなくてはならない。型付け規則 T-Let は、let の型が以下のように計算できることを述べている。まず、let 束縛項の型を計算し、その型を持つ束縛で文脈を拡張してから、拡張された文脈で本体の型を計算すれば、その型が let 式全体の型となる。

演習 11.5.1. [推奨, ★★★]　型検査器 letexercise（この本のウェブサイトで入手可能）は、let 式の不完全な実装である。基本的な構文解析と表示の関数は実装されているが、**eval1** 関数および **typeof** 関数は TmLet のための節が欠けている（それらの箇所には、すべてのものにマッチして、アサーション違反でプログラムをクラッシュさせるようなダミーの節が書かれている）。これを完成させよ。□

let も派生形式として定義できるだろうか。Landin が示した通り、可能である。ただし、その詳細は逐次実行や型指定の場合よりも若干複雑である。素朴には、ラムダ抽象と関数適用を組み合わせて使うことで let 束縛の効果を得られるのは明らかである。

$$\texttt{let } x\texttt{=}t_1 \texttt{ in } t_2 \quad \stackrel{\text{def}}{=} \quad (\lambda x{:}T_1.t_2)\ t_1$$

ただしこの略記の右辺には、左辺に含まれていない型注釈 T_1 が含まれることに注意されたい。つまり、何らかのコンパイラが構文解析フェーズで派生形式を脱糖衣するところを想像してみると、脱糖衣した内部言語の項における λ の型注釈として T_1 を生成すればよいということを、いったいどうして構文解析器が知りうるのかが問題となる。

答えは、もちろん知りえない。T_1 は型検査器からわかるのだ。t_1 の型を計算すれば必要な型注釈が見つかる。このことから、より形式的には、let 構築子はこれまで見てきた他の派生形式とは少し異なり、項に対する脱糖衣変換とはみなせず、**型付け導出**（別の言い方をすれば、型検査器の解析結果を付与した項）に対する変換とみなす必要があるということがわかる。この変換は、let に

関する導出

$$\frac{\dfrac{\vdots}{\Gamma \vdash \mathtt{t}_1 : \mathtt{T}_1} \qquad \dfrac{\vdots}{\Gamma, \mathtt{x}{:}\mathtt{T}_1 \vdash \mathtt{t}_2 : \mathtt{T}_2}}{\Gamma \vdash \mathtt{let}\ \mathtt{x}{=}\mathtt{t}_1\ \mathtt{in}\ \mathtt{t}_2 : \mathtt{T}_2}\ \text{T-LET}$$

を以下のようなラムダ抽象と関数適用を用いた導出へと写すものである。

$$\frac{\dfrac{\dfrac{\vdots}{\Gamma, \mathtt{x}{:}\mathtt{T}_1 \vdash \mathtt{t}_2 : \mathtt{T}_2}}{\Gamma \vdash \lambda \mathtt{x}{:}\mathtt{T}_1.\mathtt{t}_2 : \mathtt{T}_1 \to \mathtt{T}_2}\ \text{T-ABS} \qquad \dfrac{\vdots}{\Gamma \vdash \mathtt{t}_1 : \mathtt{T}_1}}{\Gamma \vdash (\lambda \mathtt{x}{:}\mathtt{T}_1.\mathtt{t}_2)\ \mathtt{t}_1 : \mathtt{T}_2}\ \text{T-APP}$$

したがって、`let` はこれまで見てきた他の派生形式と比べると「やや派生的ではない」といえる。`let` の評価に関する振る舞いは脱糖衣によって導けるが、型付けに関する振る舞いは内部言語に組み込まれていなければならないからである。

第 22 章では、`let` を派生形式として扱わない別の理由を説明する。Hindley-Milner の（すなわち単一化ベースの）多相性を持つ言語では、`let` は型検査器によって多相的な定義を**一般化**するよう特別に扱われる。この型付けは通常の λ 抽象と関数適用では真似ることができない。

演習 11.5.2. [★★] `let` を派生形式として定義する別の方法として、直ちに「実行」する方法、すなわち `let` $\mathtt{x}{=}\mathtt{t}_1$ `in` $\mathtt{t}_2$ を、本体を置換した $[\mathtt{x} \mapsto \mathtt{t}_1]\mathtt{t}_2$ の略記とみなす方法が考えられる。これは良いアイディアだろうか。 □

11.6 二つ組

ほとんどのプログラミング言語には複合的なデータ構造を組み立てるためのさまざまな方法がある。その中で最も単純なものは値の**二つ組**、より一般化すれば任意個の値の**組**だろう。この節では二つ組について扱い、より一般化された組やラベル付きレコードに関しては、11.7 節および 11.8 節で扱う[†3]。

二つ組の形式化はほとんど単純すぎて議論する価値もないように思われる。本書をここまで読んだならば、図 11-5 の規則を読むのは、同じ内容を自然言語で書いたものと同程度に簡単に読めるはずである。とはいうものの、共通するパターンを強調するために、この定義のいろいろな部分を手短に見ていこう。

単純型付きラムダ計算への二つ組の追加は以下の追加を伴う。二つ組化 $\{\mathtt{t}_1,\mathtt{t}_2\}$ と射影（第一要素の射影 `t.1` と第二要素の射影 `t.2`）の二つの形式の項、および $\mathtt{T}_1$ と $\mathtt{T}_2$ の**直積**もしくは**デカルト積**と呼ばれる新しい型構築子 $\mathtt{T}_1 \times \mathtt{T}_2$ である。波括弧を用いて二つ組を表記[†4]するのは、11.8 節で説明するレコードとの関係を強調するためである。

評価に関しては、二つ組や射影の振る舞いを規定する新しい規則を幾つか追加する必要がある。E-PAIRBETA1 と E-PAIRBETA2 は、完全に評価済みの二つ組に対する第一射影もしくは第二射影の結果が、対応する要素であることを規定している。E-PROJ1 および E-PROJ2 は、射影される項が完

---

†3 実装 `fullsimple` は実際にはこの節で説明する二つ組の構文を提供していない。これは結局は組のほうがより一般的だからである。

†4 この波括弧を用いた表記は、標準的な数学の記法での集合の記法を想起するため、二つ組や組に対しては若干好ましくない。ML のような人気のある言語と研究文献のいずれにおいても、二つ組や組は丸括弧で囲むのがより一般的である。角括弧や山括弧が用いられる場合もある。

$\rightarrow \times$ 　　　　　　　　　　　　　　　　　　　　　　　　　$\lambda_{\rightarrow}$（図 9-1）の拡張

新しい構文形式

t ::=	...	項：
	$\{t,t\}$	二つ組
	$t.1$	第一要素の射影
	$t.2$	第二要素の射影
v ::=	...	値：
	$\{v,v\}$	二つ組値
T ::=	...	型：
	$T_1 \times T_2$	直積型

新しい評価規則　$t \longrightarrow t'$

$$\{v_1,v_2\}.1 \longrightarrow v_1 \qquad \text{(E-PairBeta1)}$$

$$\{v_1,v_2\}.2 \longrightarrow v_2 \qquad \text{(E-PairBeta2)}$$

$$\frac{t_1 \longrightarrow t_1'}{t_1.1 \longrightarrow t_1'.1} \qquad \text{(E-Proj1)}$$

$$\frac{t_1 \longrightarrow t_1'}{t_1.2 \longrightarrow t_1'.2} \qquad \text{(E-Proj2)}$$

$$\frac{t_1 \longrightarrow t_1'}{\{t_1,t_2\} \longrightarrow \{t_1',t_2\}} \qquad \text{(E-Pair1)}$$

$$\frac{t_2 \longrightarrow t_2'}{\{v_1,t_2\} \longrightarrow \{v_1,t_2'\}} \qquad \text{(E-Pair2)}$$

新しい型付け規則　$\Gamma \vdash t : T$

$$\frac{\Gamma \vdash t_1 : T_1 \qquad \Gamma \vdash t_2 : T_2}{\Gamma \vdash \{t_1,t_2\} : T_1 \times T_2} \qquad \text{(T-Pair)}$$

$$\frac{\Gamma \vdash t_1 : T_{11} \times T_{12}}{\Gamma \vdash t_1.1 : T_{11}} \qquad \text{(T-Proj1)}$$

$$\frac{\Gamma \vdash t_1 : T_{11} \times T_{12}}{\Gamma \vdash t_1.2 : T_{12}} \qquad \text{(T-Proj2)}$$

▲ 図 11-5.　二つ組

全に評価されていない場合は、その項の簡約を進めることを許している。E-Pair1 および E-Pair2 は二つ組の要素を評価する。最初に左側を評価し、左側が値になったら右側を評価する。

これらの規則でのメタ変数 v、t の使い方から定まる順序によって、二つ組の左から右への評価戦略が強制される。例えば、複合的な項

```
{pred 4, if true then false else false}.1
```

は、以下のように（だけ）評価される。

```
    {pred 4, if true then false else false}.1
⟶  {3, if true then false else false}.1
⟶  {3, false}.1
⟶  3
```

値の定義にも新たな節を追加し、$\{v_1,v_2\}$ を値とする必要がある。二つ組の値の要素それ自体も値でなくてはならないという事実によって、関数引数として渡された二つ組は関数の本体の評価前に完全に評価されることが保証される。例えば以下のようになる。

```
    (λx:Nat×Nat. x.2) {pred 4, pred 5}
⟶  (λx:Nat×Nat. x.2) {3, pred 5}
⟶  (λx:Nat×Nat. x.2) {3,4}
⟶  {3,4}.2
⟶  4
```

$\rightarrow$ {} λ→（図 9-1）の拡張

新しい構文形式

$\mathtt{t}$	::=	...	項：
		$\{\mathtt{t}_i{}^{i\in 1..n}\}$	組
		$\mathtt{t.i}$	射影
$\mathtt{v}$	::=	...	値：
		$\{\mathtt{v}_i{}^{i\in 1..n}\}$	組の値
$\mathtt{T}$	::=	...	型：
		$\{\mathtt{T}_i{}^{i\in 1..n}\}$	組の型

新しい評価規則 $\boxed{\mathtt{t} \longrightarrow \mathtt{t}'}$

$$\{\mathtt{v}_i{}^{i\in 1..n}\}\mathtt{.j} \longrightarrow \mathtt{v}_j \qquad \text{(E-ProjTuple)}$$

$$\frac{\mathtt{t}_1 \longrightarrow \mathtt{t}_1'}{\mathtt{t}_1\mathtt{.i} \longrightarrow \mathtt{t}_1'\mathtt{.i}} \qquad \text{(E-Proj)}$$

$$\frac{\mathtt{t}_j \longrightarrow \mathtt{t}_j'}{\begin{array}{l}\{\mathtt{v}_i{}^{i\in 1..j-1}, \mathtt{t}_j, \mathtt{t}_k{}^{k\in j+1..n}\}\\ \longrightarrow \{\mathtt{v}_i{}^{i\in 1..j-1}, \mathtt{t}_j', \mathtt{t}_k{}^{k\in j+1..n}\}\end{array}} \qquad \text{(E-Tuple)}$$

新しい型付け規則 $\boxed{\Gamma \vdash \mathtt{t} : \mathtt{T}}$

$$\frac{\text{各 } i \text{ に対して } \Gamma \vdash \mathtt{t}_i : \mathtt{T}_i}{\Gamma \vdash \{\mathtt{t}_i{}^{i\in 1..n}\} : \{\mathtt{T}_i{}^{i\in 1..n}\}} \qquad \text{(T-Tuple)}$$

$$\frac{\Gamma \vdash \mathtt{t}_1 : \{\mathtt{T}_i{}^{i\in 1..n}\}}{\Gamma \vdash \mathtt{t}_1\mathtt{.j} : \mathtt{T}_j} \qquad \text{(T-Proj)}$$

▲図 11-6. 組

二つ組と射影に関する型付け規則は単純である。導入規則 T-Pair は、$\{\mathtt{t}_1,\mathtt{t}_2\}$ が型 $\mathtt{T}_1\times\mathtt{T}_2$ を持つのは、$\mathtt{t}_1$ が型 $\mathtt{T}_1$ を持ち、$\mathtt{t}_2$ が型 $\mathtt{T}_2$ を持つときであると述べている。逆に、除去規則 T-Proj1 および T-Proj2 がいっているのは、$\mathtt{t}_1$ が直積型 $\mathtt{T}_{11}\times\mathtt{T}_{12}$ を持つ（すなわち、二つ組へと評価される）ならば、その組の射影の型はそれぞれ $\mathtt{T}_{11}$ と $\mathtt{T}_{12}$ であるということである。

11.7 組

前節の二項の直積を n 項の直積に一般化することは容易である。そのような直積は**組**と呼ばれることが多い。例えば、`{1,2,true}` は二つの数と一つのブール値からなる三項の組である。この項の型は `{Nat,Nat,Bool}` と書かれる。

この一般化の唯一の問題は、体系を形式化するのに引数の数が任意である構造を一様に記述する記法を考え出さなければならないことである。そのような記法には常に少し問題がある。というのも厳密さと読みやすさとの間のバランスが難しいためである。$\mathtt{t}_1$ から $\mathtt{t}_n$ の n 個の項からなる組を $\{\mathtt{t}_i{}^{i\in 1..n}\}$ と書き、その型は $\{\mathtt{T}_i{}^{i\in 1..n}\}$ と書くものとする。ここでの n は 0 でもよいことに注意されたい。その場合、範囲 $1..n$ は空で、$\{\mathtt{t}_i{}^{i\in 1..n}\}$ は空の組 `{}` となる。また、5 のような裸の値と、`{5}` のような 1 要素の組との違いにも注意されたい。後者に対して許される操作はその第一要素の射影のみである。

組の形式化は図 11-6 の通りである。この定義は直積の定義（図 11-5）と同様である。ただし、二つ組化の規則は n 項の場合に一般化され、第一射影と第二射影のそれぞれについて別々に与えられていた規則が、組からの任意の射影に関する単一の規則に一般化されているという点が異なる。特別なコメントに値する規則は、E-Tuple だけである。この規則は、図 11-5 の規則 E-Pair1 と規則 E-Pair2 とを組み合わせ、一般化したものになっている。これは日本語で説明すると、もしフィールド j の左までの全フィールドが値に簡約済みの組があったら、フィールド j を $\mathtt{t}_j$ から $\mathtt{t}_j'$ へと 1 ステップで評価できるということである。ここでもメタ変数の使い方によって左から右への評価戦略を強制している。

$\rightarrow$ {}　　　　λ→（図 9-1）の拡張

新しい構文形式

t ::=	...		項：
	$\{l_i = t_i\ ^{i\in 1..n}\}$		レコード
	$t.l$		射影
v ::=	...		値：
	$\{l_i = v_i\ ^{i\in 1..n}\}$		レコードの値
T ::=	...		型：
	$\{l_i : T_i\ ^{i\in 1..n}\}$		レコードの型

新しい評価規則　$t \longrightarrow t'$

$$\{l_i = v_i\ ^{i\in 1..n}\}.l_j \longrightarrow v_j \qquad \text{(E-ProjRcd)}$$

$$\frac{t_1 \longrightarrow t_1'}{t_1.l \longrightarrow t_1'.l} \qquad \text{(E-Proj)}$$

$$\frac{t_j \longrightarrow t_j'}{\{l_i = v_i\ ^{i\in 1..j-1}, l_j = t_j, l_k = t_k\ ^{k\in j+1..n}\} \longrightarrow \{l_i = v_i\ ^{i\in 1..j-1}, l_j = t_j', l_k = t_k\ ^{k\in j+1..n}\}} \qquad \text{(E-Rcd)}$$

新しい型付け規則　$\Gamma \vdash t : T$

$$\frac{\text{各}\ i\ \text{に対して}\ \Gamma \vdash t_i : T_i}{\Gamma \vdash \{l_i = t_i\ ^{i\in 1..n}\} : \{l_i : T_i\ ^{i\in 1..n}\}} \qquad \text{(T-Rcd)}$$

$$\frac{\Gamma \vdash t_1 : \{l_i : T_i\ ^{i\in 1..n}\}}{\Gamma \vdash t_1.l_j : T_j} \qquad \text{(T-Proj)}$$

▲ 図 11-7.　レコード

11.8　レコード

n 項の組からラベル付きレコードへの一般化も同じくらい単純である。単に各フィールド t_i に、あらかじめ定められた集合 $\mathcal{L}$ から選んだ**ラベル** l_i を注記すればよい。例えば、{x=5} と {partno=5524,cost=30.27} はいずれもレコード値である。型はそれぞれ {x:Nat} と {partno:Nat,cost:Float} になる。あるレコードの項や型に現れるラベルはすべて相異なるとする。

レコードのための規則を図 11-7 に示す。注意する必要があるのは E-ProjRcd だけである。この規則は若干非形式的な慣習的表記を用いているが、以下のように解釈されることを意図している。もし、$\{l_i = v_i\ ^{i\in 1..n}\}$ がレコードで、l_j がその j 番目のフィールドのラベルであるとき、$\{l_i = v_i\ ^{i\in 1..n}\}.l_j$ は 1 ステップで j 番目の値 v_j へと評価される。この表記法（および E-ProjTuple で用いた同様の表記法）は、より明示的な形で規則を書き直すことによって除去可能だが、読みやすさを大きく損なう。

演習 11.8.1. [★ ↛]　比較のため、E-ProjRcd をより明示的に記述せよ。　□

組とレコードの両定義の左上角の機能リストに、同じ「機能記号」{} が現れていることに注意されたい。実際、単にラベルの集合にアルファベットの識別子だけでなく自然数も含むことを許せば、組はレコードの特殊な場合とみなせる。そして、レコードの i 番目のフィールドがラベル i を持つときは、ラベルを省略することとする。例えば、{Bool,Nat,Bool} を {1:Bool,2:Nat,3:Bool} の略記とみなすのである。（この慣習に従えば、名前によるフィールド指定と位置によるフィールド指定を混在して使える。例えば {a:Bool,Nat,c:Bool} を {a:Bool,2:Nat,c:Bool} の略記として用いることができる。もっとも、これは現実にはあまり有用ではないだろう。）実際には、多くの言語では組とレコードの表記は異なっている。これはコンパイラにおける実装が異なっているという、実際的な理由によるものである。

プログラミング言語によって、レコードのフィールドの順序の扱いは異なる。多くの言語では、

→ {} let p（型無し） 図 11-7 と図 11-4 の拡張

新しい構文形式

p	::=	x	変数パターン
		$\{l_i\text{=}p_i\ ^{i\in 1..n}\}$	レコードパターン
t	::=	...	項：
		let p=t in t	パターン束縛

パターンマッチの規則

$$match(x, v) = [x \mapsto v] \quad \text{(M-VAR)}$$

$$\frac{\text{各 } i \text{ に対して } match(p_i, v_i) = \sigma_i}{match(\{l_i\text{=}p_i\ ^{i\in 1..n}\}, \{l_i\text{=}v_i\ ^{i\in 1..n}\}) = \sigma_1 \circ \cdots \circ \sigma_n} \quad \text{(M-RCD)}$$

新しい評価規則 $\boxed{t \longrightarrow t'}$

$$\text{let } p\text{=}v_1 \text{ in } t_2 \longrightarrow match(p, v_1)\, t_2 \quad \text{(E-LETV)}$$

$$\frac{t_1 \longrightarrow t_1'}{\text{let } p\text{=}t_1 \text{ in } t_2 \longrightarrow \text{let } p\text{=}t_1' \text{ in } t_2} \quad \text{(E-LET)}$$

▲ 図 11-8. （型無し）レコードパターン

レコード値においてもレコード型においても、フィールドの順序は意味に影響しない。すなわち、{partno=5524,cost=30.27} という項と {cost=30.27,partno=5524} という項は同じ意味および同じ型を持ち、その型は {partno:Nat,cost:Float} と書いてもよいし {cost:Float, partno:Nat} と書いてもよい。しかし、本書ではもう一方の形式化を選択することにする。すなわち、{partno=5524, cost=30.27} と {cost=30.27,partno=5524} は異なったレコード値であり、それぞれ {partno:Nat,cost:Float} および {cost:Float, partno:Nat} という型を持つものとする。第 15 章では部分型関係を導入することで、順序に関してより柔軟な見方を採用する。具体的には {partno:Nat,cost:Float} と {cost:Float,partno:Nat} が**同値**、すなわちお互いが他方の部分型であるようにし、一方の型の項を他方の型が期待される任意の文脈で使えるようにする。(部分型付けがある場合には、順序付きと順序なしのレコードの選択はパフォーマンスに大きく影響する。これについては 15.6 節で詳しく述べる。しかし、いったん順序なしのレコードを用いると決めてしまえば、レコードを最初から順序なしだったと考えるのか、もしくは元々順序があったが順序を無視することを許すような規則を追加したと考えるのかは、純粋に好みの問題にすぎない。本書では、両方の変種を議論できるよう、後者のアプローチを採用する。)

演習 11.8.2. [★★★] 本書のレコードの形式化では、射影操作を用いて一度に一つずつレコードのフィールドを取り出す。多くの高水準プログラミング言語では、代わりに**パターンマッチ**の構文が提供されており、すべてのフィールドを一度に取り出せる。それによりある種のプログラムは遥かに簡潔に表現できる。パターンは通常は入れ子にもでき、複雑な入れ子データ構造の一部を容易に取り出せる。

レコードを持つ型無しラムダ計算に対して、単純な形のパターンマッチを追加するには、新たな構文要素として**パターン**を追加し、項の構文に（パターンマッチ構文自体のための）新たなケースを追加すればよい（図 11-8 参照）。

パターンマッチの計算規則は、図 11-4 の let 束縛の規則を一般化したものになっている。そこで用いられている補助的な関数は、パターン p と値 v を受け取り、失敗する（これは v が p にマッチしないことを表す）か、もしくは p 中に現れる変数を v の対応する部分に写すような代入を生成する。例えば、*match*({x,y}, {5,true}) は [x ↦ 5, y ↦ true] という代入を生成し、*match*(x, {5,true}) は [x ↦ {5,true}] という代入を生成する。一方、*match*({x}, {5,true}) は失敗する。E-LETV は *match* を用いて p 中の変数への適切な代入を計算する。

match 関数自体は別の一連の推論規則によって定義される。規則 M-VAR は、変数パターンは常に成功し、その変数をマッチされている値全体に写す代入を返す、ということを示している。規則 M-RCD は、レコードパターン $\{\mathtt{l}_i\mathtt{=p}_i\ ^{i\in 1..n}\}$ を（同じ長さで同じラベルの）レコード値 $\{\mathtt{l}_i\mathtt{=v}_i\ ^{i\in 1..n}\}$ にマッチさせるには、各部分パターン $\mathtt{p}_i$ を対応する値 $\mathtt{v}_i$ にマッチさせて代入 σ_i を得、それらすべての代入を合成することで最終結果となる代入を構成すればよい、ということを示している。（一つのパターンに同じ変数は高々一回しか出現しないことを要求しているので、この代入の合成は単に元の代入の和集合である。）

この体系に型を追加する方法を示せ。

(1) 新しい構文のための型付け規則を与えよ。（その過程で必要と感じた構文の変更を行ってよい。）
(2) 計算体系全体のための型保存と進行の証明の概略を示せ。（完全な証明は不要で、必要な補題の言明を適切な順番で示すだけでよい。） □

11.9 和

多くのプログラムは**異種の値の混在した**集まりを扱う必要がある。例えば、二分木のノードは葉もしくは二つの子を持った内部ノードである。同様に、リストのセルは `nil` もしくは、先頭の要素と残りの部分とを持つ `cons` セルである[†5]。同様に、コンパイラにおける抽象構文木のノードは、変数、ラムダ抽象、関数適用などを表しうる。この種のプログラミングをサポートする型理論的な機構が**バリアント型**である。

完全に一般化されたバリアントを（11.10 節にて）導入する前に、より単純な二項の**和型**を考えてみよう。和型はちょうど二つの与えられた型から得られる値の集合を表す。例えば、以下の型を用いて、二種類のアドレス帳のレコードを表しているとしよう。

```
PhysicalAddr = {firstlast:String, addr:String};
VirtualAddr  = {name:String, email:String};
```

両方の種類のレコードを一様に扱いたい（例えば、両方の種類のレコードを含んだリストを作りたい）ときには、以下のような和型が導入できる[†6]。

```
Addr = PhysicalAddr + VirtualAddr;
```

この型の各要素は、`PhysicalAddr` か `VirtualAddr` のいずれかである。

この型の値を作るには、構成要素の型 `PhysicalAddr` もしくは `VirtualAddr` の値を、**タグ付け**すればよい。例えば、`pa` が `PhysicalAddr` 型ならば、`inl pa` は `Addr` 型である。（タグ `inl` および `inr` の名前は以下のような関数を考えることからきている。

```
inl  :  PhysicalAddr → PhysicalAddr+VirtualAddr
inr  :  VirtualAddr → PhysicalAddr+VirtualAddr
```

この関数は、`PhysicalAddr` もしくは `VirtualAddr` の要素を、和型 `Addr` の左もしくは右の構成要素に「入射（inject）」する[†7]。ただし、我々の形式化ではこれらを関数としては**扱っていない**。）

[†5] これらの例は、実世界のバリアント型の使われ方のほとんどと同じく、**再帰型**を含んでいる。例えば、リストの尾部はそれ自体リストであるといった具合に。再帰型については第 20 章で再び取り上げる。

[†6] `fullsimple` の実装は二項の和のためのここで説明する機構をサポートしておらず、後述するより一般化されたバリアントの機構だけをサポートしている。

[†7] （訳注）inl、inr はそれぞれ inject left、inject right の略。

→ + λ→（図 9-1）の拡張

新しい構文形式

t	::=	...	項：
		inl t	タグ付け（左）
		inr t	タグ付け（右）
		case t of inl x⇒t | inr x⇒t	場合分け
v	::=	...	値：
		inl v	タグ付けの値（左）
		inr v	タグ付けの値（右）
T	::=	...	型：
		T+T	和型

新しい評価規則 $t \longrightarrow t'$

$$\texttt{case (inl } v_0\texttt{) of inl } x_1{\Rightarrow}t_1 \mid \texttt{inr } x_2{\Rightarrow}t_2 \longrightarrow [x_1 \mapsto v_0]t_1 \qquad \text{(E-CASEINL)}$$

$$\texttt{case (inr } v_0\texttt{) of inl } x_1{\Rightarrow}t_1 \mid \texttt{inr } x_2{\Rightarrow}t_2 \longrightarrow [x_2 \mapsto v_0]t_2 \qquad \text{(E-CASEINR)}$$

$$\frac{t_0 \longrightarrow t_0'}{\texttt{case } t_0 \texttt{ of inl } x_1{\Rightarrow}t_1 \mid \texttt{inr } x_2{\Rightarrow}t_2 \longrightarrow \texttt{case } t_0' \texttt{ of inl } x_1{\Rightarrow}t_1 \mid \texttt{inr } x_2{\Rightarrow}t_2} \qquad \text{(E-CASE)}$$

$$\frac{t_1 \longrightarrow t_1'}{\texttt{inl } t_1 \longrightarrow \texttt{inl } t_1'} \qquad \text{(E-INL)}$$

$$\frac{t_1 \longrightarrow t_1'}{\texttt{inr } t_1 \longrightarrow \texttt{inr } t_1'} \qquad \text{(E-INR)}$$

新しい型付け規則 $\Gamma \vdash t : T$

$$\frac{\Gamma \vdash t_1 : T_1}{\Gamma \vdash \texttt{inl } t_1 : T_1{+}T_2} \qquad \text{(T-INL)}$$

$$\frac{\Gamma \vdash t_1 : T_2}{\Gamma \vdash \texttt{inr } t_1 : T_1{+}T_2} \qquad \text{(T-INR)}$$

$$\frac{\Gamma \vdash t_0 : T_1{+}T_2 \qquad \Gamma, x_1{:}T_1 \vdash t_1 : T \qquad \Gamma, x_2{:}T_2 \vdash t_2 : T}{\Gamma \vdash \texttt{case } t_0 \texttt{ of inl } x_1{\Rightarrow}t_1 \mid \texttt{inr } x_2{\Rightarrow}t_2 : T} \qquad \text{(T-CASE)}$$

▲図 11-9. 和

一般に、T_1+T_2 型の要素は、T_1 の要素をトークン inl でタグ付けしたもの、および T_2 の要素を inr でタグ付けしたものになっている。

和型の要素を**使用**するために、与えられた値が和の左側と右側のどちらからきたのか判別を可能にする case 構文を導入する。例えば、Addr から名前を取り出すには以下のようにすればよい。

```
getName = λa:Addr.
  case a of
    inl x ⇒ x.firstlast
  | inr y ⇒ y.name;
```

引数 a が PhysicalAddr を inl でタグ付けしたものなら、この case 式は最初の分岐を選び、変数 x をその PhysicalAddr に束縛する。そしてこの分岐の本体は、x から firstlast フィールドを取り出し、それを返す。同様に、もし a が VirtualAddr の値を inr でタグ付けしたものなら、二つ目の分岐が選ばれ、その VirtualAddr の name フィールドが返される。したがって、getName 関数全体の型は Addr→String となる。

前述の直観を形式化したものが図 11-9 である。項の構文には、左右の入射および、case 構文を追加する。型には和の構築子を追加する。評価に関しては、case 構文のための二つの「ベータ簡約」規則を追加する。一方は、第一の部分項が inl でタグ付けされた値 v_0 に簡約される場合のた

めのもので、他方は inr でタグ付けされた値 v_0 に簡約される場合のためのものである。いずれの場合でも、適切な本体を選び、束縛変数を v_0 で置き換える。残る評価規則は、case の最初の部分項や、タグ inl とタグ inr の下の部分項の評価を行う規則である。

タグ付けの型付け規則は単純である。inl t_1 が和型 T_1+T_2 を持つためには、t_1 が和の左側の型 T_1 に属せばよい。inr についても同様である。case 構文については、まず最初の部分項が和型 T_1+T_2 を持つことを確認し、次に二つの分岐の本体 t_1 および t_2 が同じ結果の型 T を持つことを、束縛変数 x_1 および x_2 がそれぞれ型 T_1 および型 T_2 を持つことを仮定した上で確認する。すると、case 全体の結果は T となる。以前の定義からの慣習に従って、図 11-9 は、変数 x_1 および x_2 のスコープが分岐の本体 t_1 および t_2 であることを明示的には表していない。しかし、この事実は型付け規則 T-CASE における文脈の拡張の仕方から読み取れる。

演習 11.9.1. [★★] case の型付け規則と図 8-1 の if の型付け規則の類似性に着目せよ。if は分岐に対して情報を渡さない、case を縮退した形の一種とみなせる。この直観を形式化するため、和型と Unit を用いて true、false、if を派生形式として定義せよ。 □

和と型の一意性

純粋な $\lambda_{\rightarrow}$ の型付け関係に関する性質（9.3 節参照）のほとんどは、和を追加した体系でも保たれる。ただし、一つだけ重要な性質が成り立たなくなる。型の一意性の定理（定理 9.3.3）である。この問題はタグ付けの構文 inl と inr に起因する。例えば、型付け規則 T-INL によれば、t_1 が T_1 の要素であることを示せば、**任意の** T_2 について inl t_1 は T_1+T_2 の要素であることを導出できる。例えば、inl 5 : Nat+Nat と inl 5 : Nat+Bool は共に導出可能である（他にも無限個の型が導出可能である)。型の一意性が成り立たないことが意味しているのは、これまで見てきたすべての機能のように単純に「規則を下から上に読む」ことによって型検査アルゴリズムを作ることができない、ということである。現時点ではさまざまな選択肢がある。

(1) 型検査アルゴリズムを複雑化して、T_2 の値をうまく「推測」するようにする。具体的には、その時点では T_2 を不定としておき、後からその値が何であるべきだったかを見つける。このようなテクニックについては、型の再構築を扱う際（第 22 章）に、詳しく探索することにする。
(2) 型の言語を改良し、可能なすべての T_2 の値を統一的に表現できるようにする。この選択肢については、部分型付けについて議論する際（第 15 章）に、探索することにする。
(3) どの型 T_2 を意図しているのか、プログラマが明示的な**注釈**として書くことを要求する方法もある。この選択肢は最も単純で、かつ一見して思うほどには非現実的でもない。なぜなら、本格的な言語設計では、そのような明示的な注釈は他の言語機構としばしば「抱き合わせ」にでき、事実上表に出ないようにできるためである（以下の節でこの点について再度議論する)。今のところはこの選択肢をとる。

図 11-9 に対して必要な拡張を図 11-10 に示す。単に inl t もしくは inr t と書くのではなく、inl t as T もしくは inr t as T と書くことにする。ここで T は入射した要素が属すべき和型全体である。型付け規則 T-INL と T-INR は、入射される項が本当に和の適切な分岐に属していることを検査した後に、宣言された和型を入射全体の型として用いる。（規則中で T_1+T_2 を繰り返し書くことを避けるために、構文規則では入射の注釈として任意の型 T が現れることを許している。しかし、型付け規則によって、入射が正しく型付けされていれば、注釈は常に和型であることが保証されて

→ +　　　　λ→（図 11-9）の拡張

新しい構文形式

$$\begin{array}{lll} t ::= & \ldots & \text{項：} \\ & \texttt{inl}\ t\ \texttt{as}\ T & \text{タグ付け（左）} \\ & \texttt{inr}\ t\ \texttt{as}\ T & \text{タグ付け（右）} \end{array}$$

$$\begin{array}{lll} v ::= & \ldots & \text{値：} \\ & \texttt{inl}\ v\ \texttt{as}\ T & \text{タグ付けの値（左）} \\ & \texttt{inr}\ v\ \texttt{as}\ T & \text{タグ付けの値（右）} \end{array}$$

新しい評価規則　　$t \longrightarrow t'$

$$\begin{array}{l} \texttt{case}\ (\texttt{inl}\ v_0\ \texttt{as}\ T_0) \\ \texttt{of}\ \texttt{inl}\ x_1 \Rightarrow t_1 \mid \texttt{inr}\ x_2 \Rightarrow t_2 \\ \quad \longrightarrow [x_1 \mapsto v_0]t_1 \end{array} \quad \text{(E-CaseInl)}$$

$$\begin{array}{l} \texttt{case}\ (\texttt{inr}\ v_0\ \texttt{as}\ T_0) \\ \texttt{of}\ \texttt{inl}\ x_1 \Rightarrow t_1 \mid \texttt{inr}\ x_2 \Rightarrow t_2 \\ \quad \longrightarrow [x_2 \mapsto v_0]t_2 \end{array} \quad \text{(E-CaseInr)}$$

$$\frac{t_1 \longrightarrow t_1'}{\texttt{inl}\ t_1\ \texttt{as}\ T_2 \longrightarrow \texttt{inl}\ t_1'\ \texttt{as}\ T_2} \quad \text{(E-Inl)}$$

$$\frac{t_1 \longrightarrow t_1'}{\texttt{inr}\ t_1\ \texttt{as}\ T_2 \longrightarrow \texttt{inr}\ t_1'\ \texttt{as}\ T_2} \quad \text{(E-Inr)}$$

新しい型付け規則　　$\Gamma \vdash t : T$

$$\frac{\Gamma \vdash t_1 : T_1}{\Gamma \vdash \texttt{inl}\ t_1\ \texttt{as}\ T_1{+}T_2 : T_1{+}T_2} \quad \text{(T-Inl)}$$

$$\frac{\Gamma \vdash t_1 : T_2}{\Gamma \vdash \texttt{inr}\ t_1\ \texttt{as}\ T_1{+}T_2 : T_1{+}T_2} \quad \text{(T-Inr)}$$

▲ 図 11-10.　和（一意型付け）

いる。）型注釈の構文は 11.4 節の型指定の機構を連想させることを意図している。事実上、この注釈は構文上必須の型指定とみなせる。

11.10 バリアント

二項の和を一般化すると、ラベル付きの**バリアント**になる。これは直積がラベル付きレコードに一般化できるのと同様である。$T_1{+}T_2$ と書く代わりに、$<l_1{:}T_1, l_2{:}T_2>$ と書く。ここで、l_1 と l_2 はバリアントのラベルである。また、inl t as $T_1{+}T_2$ と書く代わりに $<l_1{=}t>$ as $<l_1{:}T_1, l_2{:}T_2>$ と書く。case における分岐に inl および inr でラベルを付ける代わりに、対応する和型と同じラベルを用いる。これらの一般化を用いると、前節の getAddr の例は以下のようになる。

```
  Addr = <physical:PhysicalAddr, virtual:VirtualAddr>;
  a = <physical=pa> as Addr;
▸ a : Addr

  getName = λa:Addr.
    case a of
      <physical=x> ⇒ x.firstlast
    | <virtual=y> ⇒ y.name;
▸ getName : Addr → String
```

バリアントの形式的定義を図 11-11 で与える。11.8 節でのレコードの場合と同様に、バリアント型でのラベルの順序はここでは意味があることに注意されたい。

$\rightarrow$ <>　　　　$\lambda_{\rightarrow}$（図 9-1）の拡張

新しい構文形式

$\mathtt{t}$	::=	...	項：
		$\mathtt{<l=t>\ as\ T}$	タグ付け
		$\mathtt{case\ t\ of\ <l_i=x_i>\Rightarrow t_i}\ ^{i\in 1..n}$	場合分け
$\mathtt{v}$	::=	...	値：
		$\mathtt{<l=v>\ as\ T}$	タグ付けの値
$\mathtt{T}$	::=	...	型：
		$\mathtt{<l_i:T_i}\ ^{i\in 1..n}\mathtt{>}$	バリアント型

新しい評価規則　　$\boxed{\mathtt{t} \longrightarrow \mathtt{t'}}$

$$\mathtt{case\ (<l_j=v_j>\ as\ T)\ of\ <l_i=x_i>\Rightarrow t_i}\ ^{i\in 1..n} \longrightarrow [\mathtt{x_j} \mapsto \mathtt{v_j}]\mathtt{t_j} \qquad \text{(E-CaseVariant)}$$

$$\frac{\mathtt{t_0} \longrightarrow \mathtt{t_0'}}{\mathtt{case\ t_0\ of\ <l_i=x_i>\Rightarrow t_i}\ ^{i\in 1..n} \longrightarrow \mathtt{case\ t_0'\ of\ <l_i=x_i>\Rightarrow t_i}\ ^{i\in 1..n}} \qquad \text{(E-Case)}$$

$$\frac{\mathtt{t_i} \longrightarrow \mathtt{t_i'}}{\mathtt{<l_i=t_i>\ as\ T} \longrightarrow \mathtt{<l_i=t_i'>\ as\ T}} \qquad \text{(E-Variant)}$$

新しい型付け規則　　$\boxed{\Gamma \vdash \mathtt{t} : \mathtt{T}}$

$$\frac{\Gamma \vdash \mathtt{t_j} : \mathtt{T_j}}{\Gamma \vdash \mathtt{<l_j=t_j>\ as\ <l_i:T_i}\ ^{i\in 1..n}\mathtt{>} : \mathtt{<l_i:T_i}\ ^{i\in 1..n}\mathtt{>}} \qquad \text{(T-Variant)}$$

$$\frac{\Gamma \vdash \mathtt{t_0} : \mathtt{<l_i:T_i}\ ^{i\in 1..n}\mathtt{>} \qquad \text{各 } i \text{ に対して } \Gamma, \mathtt{x_i}:\mathtt{T_i} \vdash \mathtt{t_i} : \mathtt{T}}{\Gamma \vdash \mathtt{case\ t_0\ of\ <l_i=x_i>\Rightarrow t_i}\ ^{i\in 1..n} : \mathtt{T}} \qquad \text{(T-Case)}$$

▲図 11-11.　バリアント

オプション

バリアントに関する非常に有用なイディオムの一つに、**オプション値**がある。例えば

```
OptionalNat = <none:Unit, some:Nat>;
```

という型の要素は、自明な値 unit に none タグを付けたものか、もしくは数に some タグを付けたものかである。別の言い方をすれば、型 OptionalNat は Nat に特別な値 none を追加して拡張したものと同型である。例えば

```
Table = Nat→OptionalNat;
```

という型は、数から数への部分写像を表現する。この写像の定義域は、結果が <some=*n*> となるような *n* が存在する入力の集合である。空のテーブル

```
emptyTable = λn:Nat. <none=unit> as OptionalNat;
▸ emptyTable : Table
```

はすべての入力に対して none を返す定数関数である。構築子

```
extendTable =
  λt:Table. λm:Nat. λv:Nat.
    λn:Nat.
      if equal n m then <some=v> as OptionalNat
      else t n;
▸ extendTable : Table → Nat → Nat → Table
```

はテーブルを受け取り、入力 m を出力 <some=v> に写すエントリを追加（もしくは上書き）する。（この equal 関数は 404 ページにある演習 11.11.1 の解答で定義している。）

Table を検索して得られた結果は case でラップすることにより使える。例えば、t がテーブルで、5 のエントリを検索したいならば、以下のように書ける。

```
x = case t(5) of
      <none=u> ⇒ 999
    | <some=v> ⇒ v;
```

ただし、t が 5 に対して未定義な場合は、x の既定値として 999 を返すとする。

多くの言語にはオプションのためのサポートが組み込まれている。例えば、OCaml では型構築子 option が定義済みであり、典型的な OCaml プログラムの多くの関数がオプションを返す。同様に、C、C++、Java といった言語における null 値も、実際には姿を変えたオプションである。これら言語における T 型の変数（ただし、T は参照型、すなわちヒープに割り当てられる何かとする）は実際には、特別な値 null もしくは、T の値へのポインタを格納できる。すなわち、そのような変数の型は、実際には Option(T) = <none:Unit,some:T> としたときの Ref(Option(T)) である。Ref 構築子については第 13 章で詳しく議論する。

列挙型

特別な言及に値するくらい十分に有用な、バリアント型の「縮退した場合」が二つある。列挙型および、単一のフィールドのバリアントである。

列挙型は、各ラベルに対応するフィールドの型が Unit 型であるようなバリアント型である。例えば、平日の曜日を表す型は以下のように定義できる。

```
Weekday = <monday:Unit, tuesday:Unit, wednesday:Unit,
           thursday:Unit, friday:Unit>;
```

この型の要素は <monday=unit> as Weekday といった項である。実際、型 Unit のただ一つの要素は unit なので、型 Weekday は平日の曜日に一対一対応するちょうど五つの値を持つ。case 構文を用いて列挙型上の計算を定義できる。

```
nextBusinessDay = λw:Weekday.
  case w of <monday=x>    ⇒ <tuesday=unit> as Weekday
          | <tuesday=x>   ⇒ <wednesday=unit> as Weekday
          | <wednesday=x> ⇒ <thursday=unit> as Weekday
          | <thursday=x>  ⇒ <friday=unit> as Weekday
          | <friday=x>    ⇒ <monday=unit> as Weekday;
```

明らかに、ここで用いている具象構文は、こういったプログラムを読み書きしやすくチューニングされたものではない。一部の言語では列挙型の宣言と使用のための特別な構文を用意している（Pascal がその最初の言語である）。ML（107 ページ参照）などの他の言語では、列挙型をバリアントの特別な場合としている。

単一フィールドのバリアント

もう一つの興味深い特殊な場合は、単一のラベル l だけのバリアント型である。

```
V = <l:T>;
```

そんな型は一見するとあまり有用ではないように思えるかもしれない。というのは、V の要素はすべて、ある t : T に対し <l=t> という形のものだけのため、V の要素はフィールドの型 T の要素と一対一対応が付くからである。しかし、重要なことは、まずバリアントをアンパッケージ化しないことには、通常の T 上の演算を V の要素に適用**できない**ことである。すなわち、V を誤って T と取

り違えるおそれがない。

例えば、複数通貨のかかわる金融計算を行うプログラムを書いているとしよう。そのようなプログラムはドルとユーロを換算する関数を含んでいるかもしれない。もし、ドルとユーロの両方が Float で表現されているとしたら、それらの関数は以下のようになるだろう。

```
   dollars2euros = λd:Float. timesfloat d 1.1325;
 ▸ dollars2euros : Float → Float

   euros2dollars = λe:Float. timesfloat e 0.883;
 ▸ euros2dollars : Float → Float
```

(ここで timesfloat : Float→Float→Float は浮動小数点数の乗算である。) ドル建ての

```
   mybankbalance = 39.50;
```

から始めて、ユーロへ換算し、それをドルに戻すのは以下のようになる。

```
   euros2dollars (dollars2euros mybankbalance);
 ▸ 39.49990125 : Float
```

これらはすべて完全に妥当である。しかし、全く意味をなさない操作も同じくらい簡単にできてしまう。例えば、銀行口座の残高をユーロへと二度換算することもできてしまう。

```
   dollars2euros (dollars2euros mybankbalance);
 ▸ 50.660971875 : Float
```

すべて量が単に浮動小数点数として表現されているため、型システムを利用してこの種の無意味な操作を防ぐ方法がないのである。しかし、ドルとユーロを（浮動小数点数を実際の表現とする）異なったバリアント型とすると

```
   DollarAmount = <dollars:Float>;
   EuroAmount = <euros:Float>;
```

正しい通貨の量しか受け付けない安全なバージョンの換算関数を定義できる。

```
   dollars2euros =
     λd:DollarAmount.
       case d of <dollars=x> ⇒
         <euros = timesfloat x 1.1325> as EuroAmount;
 ▸ dollars2euros : DollarAmount → EuroAmount

   euros2dollars =
     λe:EuroAmount.
       case e of <euros=x> ⇒
         <dollars = timesfloat x 0.883> as DollarAmount;
 ▸ euros2dollars : EuroAmount → DollarAmount
```

そうすれば、型検査器は計算中で使われている通貨を把握でき、最終結果をどの通貨で解釈すべきか教えてくれる。

```
   mybankbalance = <dollars=39.50> as DollarAmount;
   euros2dollars (dollars2euros mybankbalance);
 ▸ <dollars=39.49990125> as DollarAmount : DollarAmount
```

さらに、無意味な二重換算を書いたときには、型が一致せず、そのプログラムは（正しく）拒絶される。

```
  dollars2euros (dollars2euros mybankbalance);
▸ Error: parameter type mismatch
```

バリアント 対 データ型

$<l_i : T_i \; ^{i \in 1..n}>$ という形のバリアント型 T は、だいたい以下の ML のデータ型に相当する[†8]。

```
type T = l₁ of T₁
       | l₂ of T₂
       | ...
       | lₙ of Tₙ
```

しかし、幾つかの違いについては言及しておく必要がある。

(1) つまらないことだが混乱を招く可能性がある点として、本章で仮定している識別子の大文字・小文字の慣習が OCaml と異なっていることがある。OCaml では型は小文字で始まり、データ型の構築子（本章の用語ではラベル）は大文字で始まらなくてはならない。そのため、厳密に言えば、上述のデータ型の宣言は以下のように書かなくてはならない。

```
type t = L₁ of t₁ | ... | Lₙ of tₙ
```

しかし、項 t と型 T の混同を避けるため、以降の議論では OCaml の慣習を無視し、我々の慣習を用いることにする。

(2) 最も興味深い違いは、OCaml では構築子 l_i を使って T_i の要素をデータ型 T に入射する際に型注釈が不要な点である。OCaml では単に l_i(t) と書けばよい。OCaml が（型付けの一意性を保ちつつ）注釈をなくしている方法は、データ型 T は使う前に宣言しておかなくてはならないというものである。さらに、T のラベルは同じスコープで宣言される他のデータ型で用いることができない。そのため、型検査器が l_i(t) を見たときには、型注釈としては T しかありえないことがわかる。実質的には、型注釈はラベルそのものの中に「隠されている」のである。
このトリックによって多くの馬鹿げた注釈が不要になるが、一方でユーザの一定の不平のもとにもなっている。これは異なったデータ型の間でラベルを共有できないためである（少なくとも同じモジュール内では）。第 15 章では、注釈を省きつつこの欠点を避ける別の方法を説明する。

(3) OCaml で使われている別の便利なトリックとして、データ型の定義においてラベルに対応する型が単に Unit であるときは省略してよいというものがある。これによって列挙型は例えば、

```
type Weekday = monday of Unit
             | tuesday of Unit
             | wednesday of Unit
             | thursday of Unit
             | friday of Unit
```

と書く代わりに、

```
type Weekday = monday | tuesday | wednesday | thursday | friday
```

と定義できるようになる。同様に、ラベル monday を自明な値 unit に適用したものではな

[†8] この節では、データ型の具象構文として OCaml のものを用いる。実装に関する本書の他の章との整合性のためである。しかし、この構文は ML の初期の方言に由来し、Standard ML や、ML の親類である Haskell で、本質的に同様の構文を見つけられる。データ型とパターンマッチは、日々のプログラミングにおいて、これらの言語の最も有用な特長の一つといえるだろう。

く、ラベル monday それ自体が Weekday 型の値とみなされる。

(4) 最後に、OCaml のデータ型は実際にはバリアント型と他の幾つかの機能をまとめたものになっている。それらの機能については後の章で個別に見ていく。

- データ型の定義は**再帰的**、すなわち、定義しようとしている型が定義の本体に現れてもよい。例えば、Nat のリストの標準的な定義では、cons でタグ付けされる値は二つ組であり、その第二要素は NatList である。

```
type NatList = nil
             | cons of Nat * NatList
```

- OCaml のデータ型は型変数で**パラメータ化**できる。以下の List データ型の一般的定義はその例である。

```
type 'a List = nil
             | cons of 'a * 'a List
```

  型理論的には、List は**型演算子**と呼ばれる、一種の関数として見ることができる。List は Nat から NatList など、与えられた 'a を具体的なデータ型に写すのである。型演算子は第 29 章の主題である。

直和としてのバリアント

和型やバリアント型は**直和**（互いに素な合併）と呼ばれることがある。型 T_1+T_2 は、その要素が T_1 と T_2 の要素をすべて含んでいるという意味で、T_1 と T_2 の「合併」（和集合）である。この合併は、T_1 と T_2 の要素を合わせる前に、それぞれ inl と inr でタグ付けされ、和集合の各要素が T_1 と T_2 のいずれからきたのか常に明確なため、互いに素である。**合併型**というフレーズは、**タグ無し**の（互いに素でない）合併型を表すためにも用いられる。こちらについては 15.7 節で述べる。

Dynamic 型

静的型付き言語であっても、型がコンパイル時にわからないようなデータを扱う必要に迫られることは多い。特に、データが複数の計算機や複数回のコンパイルにまたがる場合には、そのようなことが多い。例えば、データが外部のファイルシステムやデータベースに格納されたり、ネットワーク越しにやり取りされる場合である。こういった場合を安全に扱うために、多くの言語では値の型を実行時に調査する方法を提供している。

これを実現する魅力的な方法の一つは、値 v と型タグ T（ただし v は型 T を持つとする）の組を値とする、Dynamic という型を導入することである。Dynamic のインスタンスは、明示的なタグ付け機構によって作られ、型安全な typecase 機構によって調べられる。実質的に、Dynamic は型をラベルとする無限の直和と考えられる。Gordon [182]、Mycroft [330]、Abadi, Cardelli, Pierce, and Plotkin [7]、Leroy and Mauny [281]、Abadi, Cardelli, Pierce, and Rémy [8]、Henglein [203] を参照せよ。

11.11 一般的再帰

ほとんどのプログラミング言語に見られる別の機能として、再帰関数を定義する能力がある。以前（第 5 章、49 ページ）見たように、型無しラムダ計算では、fix コンビネータの助けを借りて再帰関数を定義できた。

型付きの場合も再帰関数を同様にして定義できる。例えば、以下に示す関数 iseven は、引数が偶数のときに true を返し、それ以外のときに false を返す。

```
  ff = λie:Nat→Bool.
         λx:Nat.
           if iszero x then true
           else if iszero (pred x) then false
           else ie (pred (pred x));
▶ ff : (Nat→Bool) → Nat → Bool

  iseven = fix ff;
▶ iseven : Nat → Bool

  iseven 7;
▶ false : Bool
```

背後にある直観は、fix に渡される高階関数 ff は iseven 関数の**生成子**であるというものである。ff が iseven に期待される振る舞いをある数 n まで近似した関数（すなわち、n 以下の入力に対して正しい結果を返す関数）ie に適用されると、ff は iseven のより良い近似（$n+2$ までの入力に対して正しい値を返す関数）を返す。fix を生成子に適用すると、生成子の不動点、すなわちすべての入力 n に対して期待される振る舞いをする関数が返される。

しかし、型無しの場合とは重要な違いが一つある。型付きラムダ計算内では fix それ自体を定義できないのである。実際、第 12 章では、停止しない計算に陥ることのある式は、単純型だけを用いては型付けできないことを確かめる[†9]。そこで、fix を言語内で定義する代わりに、単に新たなプリミティブとして追加し、型無しの場合の fix コンビネータの振る舞いを模倣する評価規則と、意図した使い方を捉えた型付け規則とを追加する。これらの規則を書き下したのが図 11-12 である。（略記法 letrec については後で論じる。）

数と不動点を持つ型付きラムダ計算は長い間プログラミング言語研究者のお気に入りの実験対象だった。というのは、これが**完全抽象**（Plotkin [385]; Hyland and Ong [230]; Abramsky, Jagadeesan, and Malacaria [12]）といった意味論上のさまざまな微妙な現象が起こる、最も単純な言語であるためである。この言語はしばしば **PCF** と呼ばれる。

演習 11.11.1. [★★] equal、plus、times、および factorial を fix を用いて定義せよ。 □

fix 構文は典型的には（関数から関数への関数の不動点としての）関数を組み立てるのに使われる。しかし、規則 T-Fix 中の型 T_1 が関数型に制限されていない点に注目してほしい。この余分な表現力が役立つときもある。例えば、相互再帰的な関数の**レコード**を、（関数の）レコード上の関数の不動点として定義できる。以下の iseven の実装では補助的な関数 isodd を使っている。この二つの関数はレコードのフィールドとして定義されており、このレコードの定義を抽象化したレコード ieio の要素を iseven フィールドおよび isodd フィールドの本体での再帰呼び出しに用いている。

[†9] 後の章（第 13 章および第 20 章）では、単純型の拡張の幾つかが fix を体系内で定義する能力を取り戻すことを見る。

→ fix　　　　λ→（図 9-1）の拡張

新しい構文形式

t ::= ...　　　　項：
　　fix t　　　　t の不動点

新しい評価規則　　$t \longrightarrow t'$

$$\texttt{fix}\ (\lambda x{:}T_1.t_2) \longrightarrow [x \mapsto (\texttt{fix}\ (\lambda x{:}T_1.t_2))]t_2 \quad \text{(E-FixBeta)}$$

$$\frac{t_1 \longrightarrow t_1'}{\texttt{fix}\ t_1 \longrightarrow \texttt{fix}\ t_1'} \quad \text{(E-Fix)}$$

新しい型付け規則　　$\Gamma \vdash t : T$

$$\frac{\Gamma \vdash t_1 : T_1 \to T_1}{\Gamma \vdash \texttt{fix}\ t_1 : T_1} \quad \text{(T-Fix)}$$

新しい派生形式

$$\texttt{letrec}\ x{:}T_1 = t_1\ \texttt{in}\ t_2 \stackrel{\text{def}}{=} \texttt{let}\ x = \texttt{fix}\ (\lambda x{:}T_1.t_1)\ \texttt{in}\ t_2$$

▲ 図 11-12.　一般的再帰

```
ff = λieio:{iseven:Nat→Bool, isodd:Nat→Bool}.
       {iseven = λx:Nat.
                   if iszero x then true
                   else ieio.isodd (pred x),
        isodd = λx:Nat.
                   if iszero x then false
                   else ieio.iseven (pred x)};
▸ ff : {iseven:Nat→Bool,isodd:Nat→Bool} →
       {iseven:Nat→Bool, isodd:Nat→Bool}
```

関数 ff の不動点を取ると、二つの関数からなるレコードが得られる。

```
r = fix ff;
▸ r : {iseven:Nat→Bool, isodd:Nat→Bool}
```

そして、第一要素を射影すれば iseven 関数そのものを得られる。

```
iseven = r.iseven;
▸ iseven : Nat → Bool

iseven 7;
▸ false : Bool
```

任意の T に対して、T→T 型の関数の不動点を取れることは、幾つかの驚くべき結果をもたらす。特に、**任意の**型に対して、その型に属する項が存在することを意味している。これを確認するには、任意の型 T に対して関数 $\texttt{diverge}_T$ を以下のように定義できることに注目せよ。

```
divergeT = λ_:Unit. fix (λx:T.x);
▸ divergeT : Unit → T
```

$\texttt{diverge}_T$ が unit 引数に適用されると、停止しない評価列が得られる。この列は E-FixBeta が繰り返し適用され、常に同じ項を返すような列である。つまり、任意の型 T に対して、項 $\texttt{diverge}_T$ unit は T の**未定義要素**なのである。

考えられる最後の改良の一つは、再帰的定義の結果を変数に束縛するという、よくある場合のための、より便利な具象構文を導入することである。高水準言語のほとんどでは、上でした iseven の最初の定義は、以下のように書ける。

```
letrec iseven : Nat→Bool =
  λx:Nat.
    if iszero x then true
    else if iszero (pred x) then false
    else iseven (pred (pred x))
in
  iseven 7;
▶ false : Bool
```

再帰的束縛の構文 letrec は派生形式として容易に定義できる。

$$\texttt{letrec x:}T_1\texttt{=}t_1\texttt{ in }t_2 \stackrel{\text{def}}{=} \texttt{let x = fix (}\lambda\texttt{x:}T_1\texttt{.}t_1\texttt{) in }t_2$$

演習 11.11.2. [★] 演習 11.11.1 での plus、times、factorial の定義を、fix の代わりに letrec を使って書き直せ。 □

不動点演算子に関するより詳しい情報は、Klop [264] および Winskel [496] を参照せよ。

11.12 リスト

これまで見てきた型付けの機能は、Bool や Unit といった**基本型**もしくは、既にある型から新たな型を作る → や × といった**型構築子**に分類できる。別の便利な型構築子としては List がある[†10]。任意の型 T に対して、型 List T は T から得られた要素からなる有限長のリストを表している。

リストの構文、意味論、型付けを図 11-13 に要約する。構文的な違い（T list の代わりに List T を使っているなど）と、我々の表現法ではすべての構文で明示的な型注釈を用いていることを除いて、このリストは ML や他の関数型言語に見られるものと同じである。（要素が型 T の）空リストは nil[T] と書く。リスト t_2 の先頭に、新しい（型 T の）要素 t_1 を加えて得られるリストは、cons[T] t_1 t_2 と書く。リスト t の先頭の要素（head）と残りの部分（tail）はそれぞれ head[T] t および tail[T] t と書く。ブール値を返す述語 isnil[T] t は、t が空のとき、かつそのときに限り true を返す[†11]。

演習 11.12.1. [★★★] ブール値とリストで拡張した単純型付きラムダ計算で進行定理と型保存定理が成り立つことを確かめよ。 □

演習 11.12.2. [★★] このリストの表し方は、型付け規則が文脈から容易に導出できるような注釈も含んでおり、実際には必要ない型注釈を含んでいるといえる（これらの型注釈を用いたのは、275 ページのリストの表現との比較を容易にするためである）。これら**すべて**の型注釈を除けるだろうか。 □

---

†10 （訳注：原著正誤表より）リストは fullsimple 検査器には実装されていない。

†11 ここでは単純さのために、リストの「head/tail/isnil による表現」を採用した。言語設計の観点から言えば、リストをデータ型として扱い、分解には case 式を用いるほうがよいといえる。そのほうが多くのプログラミングの誤りを型エラーとして見つけられるからである。

→ 𝔹 List　　λ→（図 9-1）のブール値（図 8-1）を用いた拡張

新しい構文形式

t ::=	...		項：
	nil[T]		空のリスト
	cons[T] t t		リスト構築子（constructor）
	isnil[T] t		リストが空かどうかの検査
	head[T] t		リストの先頭の要素（head）
	tail[T] t		リストの残りの部分（tail）
v ::=	...		値：
	nil[T]		空のリスト
	cons[T] v v		リスト構築子
T ::=	...		型：
	List T		リストの型

新しい評価規則　　$\boxed{t \longrightarrow t'}$

$$\frac{t_1 \longrightarrow t_1'}{\texttt{cons[T]}\ t_1\ t_2 \longrightarrow \texttt{cons[T]}\ t_1'\ t_2} \quad \text{(E-Cons1)}$$

$$\frac{t_2 \longrightarrow t_2'}{\texttt{cons[T]}\ v_1\ t_2 \longrightarrow \texttt{cons[T]}\ v_1\ t_2'} \quad \text{(E-Cons2)}$$

$$\texttt{isnil[S] (nil[T])} \longrightarrow \texttt{true} \quad \text{(E-IsnilNil)}$$

$$\texttt{isnil[S] (cons[T]}\ v_1\ v_2\texttt{)} \longrightarrow \texttt{false} \quad \text{(E-IsnilCons)}$$

$$\frac{t_1 \longrightarrow t_1'}{\texttt{isnil[T]}\ t_1 \longrightarrow \texttt{isnil[T]}\ t_1'} \quad \text{(E-Isnil)}$$

$$\texttt{head[S] (cons[T]}\ v_1\ v_2\texttt{)} \longrightarrow v_1 \quad \text{(E-HeadCons)}$$

$$\frac{t_1 \longrightarrow t_1'}{\texttt{head[T]}\ t_1 \longrightarrow \texttt{head[T]}\ t_1'} \quad \text{(E-Head)}$$

$$\texttt{tail[S] (cons[T]}\ v_1\ v_2\texttt{)} \longrightarrow v_2 \quad \text{(E-TailCons)}$$

$$\frac{t_1 \longrightarrow t_1'}{\texttt{tail[T]}\ t_1 \longrightarrow \texttt{tail[T]}\ t_1'} \quad \text{(E-Tail)}$$

新しい型付け規則　　$\boxed{\Gamma \vdash t : T}$

$$\Gamma \vdash \texttt{nil[}T_1\texttt{]} : \texttt{List}\ T_1 \quad \text{(T-Nil)}$$

$$\frac{\Gamma \vdash t_1 : T_1 \qquad \Gamma \vdash t_2 : \texttt{List}\ T_1}{\Gamma \vdash \texttt{cons[}T_1\texttt{]}\ t_1\ t_2 : \texttt{List}\ T_1} \quad \text{(T-Cons)}$$

$$\frac{\Gamma \vdash t_1 : \texttt{List}\ T_{11}}{\Gamma \vdash \texttt{isnil[}T_{11}\texttt{]}\ t_1 : \texttt{Bool}} \quad \text{(T-Isnil)}$$

$$\frac{\Gamma \vdash t_1 : \texttt{List}\ T_{11}}{\Gamma \vdash \texttt{head[}T_{11}\texttt{]}\ t_1 : T_{11}} \quad \text{(T-Head)}$$

$$\frac{\Gamma \vdash t_1 : \texttt{List}\ T_{11}}{\Gamma \vdash \texttt{tail[}T_{11}\texttt{]}\ t_1 : \texttt{List}\ T_{11}} \quad \text{(T-Tail)}$$

▲図 11-13.　リスト

第12章

正規化

本章で扱う体系は単純型付きラムダ計算（図 9-1）で、単一の基本型 A （図 11-1）があるとする。

この章では、純粋単純型付きラムダ計算の、もう一つの基礎的な理論的性質について考える。正しく型付けされたプログラムの評価は有限ステップで停止することが保証されている、つまり、すべての正しく型付けされた項は**正規化可能**である、という性質である。

これまで考えてきた型安全性と違い、正規化可能性は本格的なプログラミング言語にまで一般化することはできない。そういう言語はほとんどの場合、単純型付きラムダ計算を拡張し、一般的再帰（11.11 節）や再帰型（第 20 章）といった、停止しないプログラムも書ける機能を追加しているからである。しかし、363 ページで System F_ω のメタ理論を議論する際、正規化の問題が**型**のレベルで再登場する。その体系では、型レベルの言語に単純型付きラムダ計算と同じものが事実上含まれており、その型検査アルゴリズムが停止する条件は、型式の上の「正規化」操作の停止性が保証されていることに依存している。

正規化の証明を学習するもう一つの理由は、型理論の文献に見られる最も美しく、また驚くべき数学的技法の一つだからである。その中では、しばしば（この章のように）**論理関係**という基本的な証明法が用いられる。

読者によっては、最初に本書を読む際は本章を飛ばしたほうがよいかもしれない。飛ばしても以降の章を読むときに問題はない。（章間の依存関係は ix ページ参照。）

12.1 単純型の正規化

ここで考える計算体系は単一の基本型 A を持つ単純型付きラムダ計算である。この計算における正規化の証明はすべて自明というわけではない。項の簡約によって部分項中の簡約基が複製されてしまうことがあるからである。

演習 12.1.1. [★]　正しく型付けされた項のサイズに関する単純な帰納法で正規化を証明しようとすると、どこで失敗するだろうか。 □

ここで鍵となるのは（多くの帰納法による証明と同様）十分に強い帰納法の仮定を見つけることである。この目的を達成するため、各型 T に対し、型 T を持つ閉じた項からなる、ある集合 R_T を定義するところから始める。これらの集合を述語とみなし、$\mathsf{t} \in R_\mathsf{T}$ であることを $R_\mathsf{T}(\mathsf{t})$ と書く[†1]。

定義 12.1.2.

- $R_\mathsf{A}(\mathsf{t})$ であるのは、t が停止するとき、かつそのときのみである。
- $R_{\mathsf{T_1}\to\mathsf{T_2}}(\mathsf{t})$ であるのは、t が停止し、さらに、$R_{\mathsf{T_1}}(\mathsf{s})$ ならば常に $R_{\mathsf{T_2}}(\mathsf{t\ s})$ であるとき、かつそのときのみである。 □

[†1] 集合 R_T のことを**飽和集合**あるいは**簡約可能候補**と呼ぶこともある。

この定義によって、我々が必要とする、十分に強い帰納法の仮定が得られる。我々の主要な目標は、すべての**プログラム**、すなわち基本型のすべての閉じた項が停止することを示すことにある。しかし基本型の閉じた項には関数型の部分項が含まれる可能性があり、それらについても同様に、何かしら知る必要がある。さらに、これら部分項が停止するだけでは十分ではない。正規化された関数を正規化された引数に適用すると代入が発生し、これによってさらなる評価ステップが発生しうるからである。よって関数型の項に対しては、より強い条件が必要になる。すなわち、それら自身が停止すべきというだけでなく、停止する引数に適用されたときには、停止する結果を返すべきである。

定義 12.1.2 の形は、**論理関係**（ここでは単項関係のみ扱っているので、**論理述語**と呼ぶのがより適切だろう）という証明法の特徴である。もし型 A のすべての閉じた項に関する性質 P を証明したい場合、型の上での帰納法を用い、型 A のすべての項が性質 P を**満たす**ことを示し、型 A→A のすべての項が性質 P を**保存する**ことを示し、型 (A→A)→(A→A) のすべての項が、性質 P を**保存するという性質を保存する**ことを示し、などと進める。これをするために、型で添字付けられた述語の族を定義する。基本型 A の場合、述語は単に P である。関数型の場合の述語は「その関数は、入力の型における述語を満たす値を、出力の型に対する述語を満たす値に写す」である。

この定義を用いて、正規化の証明を 2 ステップで行う。まず、すべての集合 R_T のすべての要素が正規化可能であることを見る。続いて、正しく型付けされた、型 T を持つ項がすべて R_T の要素であることを示す。

最初のステップは、R_T の定義からすぐに導ける。

補題 12.1.3.　$R_T(t)$ ならば、t は停止する。　□

次のステップは二つの補題に分割される。一つ目に、R_T に所属することは評価しても不変であることを述べる。

補題 12.1.4.　t : T かつ t ⟶ t′ ならば、$R_T(t)$ は $R_T(t')$ と同値である。　□

[証明]　型 T の構造に関する帰納法による。ここでまず、t の停止性と t′ のそれが明らかに同値であることに注意されたい。もし T = A ならば、それ以上示すべきことはない。一方、ある T_1 と T_2 について $T = T_1 \to T_2$ である場合を考える。十分性（⟹）の証明のために、$R_T(t)$ であり、かつ、適当な $s : T_1$ について $R_{T_1}(s)$ であると仮定する。定義から $R_{T_2}(t\ s)$ である。しかし $t\ s \longrightarrow t'\ s$ であるから、型 T_2 についての帰納法の仮定より $R_{T_2}(t'\ s)$ が得られる。これは任意の s について成り立つから、R_T の定義より $R_T(t')$ が得られる。必要性（⟸）についても同様である。　□

次に、型 T のすべての項が R_T に属することを証明したい。ここで、帰納法は型付け導出に関するものを使う（正しく型付けされた項に関する証明で、型付け導出に関する帰納法をどこにも使わないとしたら、むしろ驚きだろう）。ここで技術的に難しいのは唯一、λ 抽象の場合の扱いである。帰納法で議論しているので、項 $\lambda x{:}T_1.t_2$ が $R_{T_1 \to T_2}$ に属することを示すには、帰納法の仮定を用いて t_2 が R_{T_2} に属することを示さなければならない。しかし R_{T_2} は**閉じた**項の集合として定義されており、一方で t_2 は x を自由変数として持ちうるので、これはうまくいかない。

この問題は、帰納法の仮定を適切に一般化するという標準的な技法を用いることで解決される。閉じた項に関する言明を証明する代わりに、それがある開いた項の閉じた**インスタンス**をすべて含むように一般化する。

補題 12.1.5.　もし $x_1{:}T_1, \ldots, x_n{:}T_n \vdash t : T$ であり、かつ $v_1, \ldots, v_n$ が型 $T_1 \ldots T_n$ の閉じた値で、さらに任意の i において $R_{T_i}(v_i)$ であるとき、$R_T([x_1 \mapsto v_1] \cdots [x_n \mapsto v_n]t)$ が成り立つ。　□

[証明] $x_1{:}T_1, \ldots, x_n{:}T_n \vdash t : T$ の導出に関する帰納法による。(最も興味深いのは、抽象の場合である。)

T-VAR の場合 $t = x_i \quad T = T_i$

直ちに明らか。

T-ABS の場合 $t = \lambda x{:}S_1.s_2 \quad x_1{:}T_1, \ldots, x_n{:}T_n, x{:}S_1 \vdash s_2 : S_2$

$T = S_1 \to S_2$

$[x_1 \mapsto v_1]\cdots[x_n \mapsto v_n]t$ は既に値であるので、値に評価されるのは明らかである。他に示さなければならないのは、$R_{S_1}(s)$ を満たす任意の $s : S_1$ において、$R_{S_2}(([x_1 \mapsto v_1]\cdots[x_n \mapsto v_n]t)\ s)$ が成り立つことである。よって、s がそのような項であると仮定する。補題 12.1.3 から、$s \longrightarrow^* v$ を満たす v が存在する。補題 12.1.4 から、$R_{S_1}(v)$ である。よって、帰納法の仮定より $R_{S_2}([x_1 \mapsto v_1]\cdots[x_n \mapsto v_n][x \mapsto v]s_2)$ がいえる。一方

$$(\lambda x{:}S_1.\ [x_1 \mapsto v_1]\cdots[x_n \mapsto v_n]s_2)\ s$$
$$\longrightarrow^* \quad [x_1 \mapsto v_1]\cdots[x_n \mapsto v_n][x \mapsto v]s_2$$

であるから、補題 12.1.4 より

$$R_{S_2}((\lambda x{:}S_1.\ [x_1 \mapsto v_1]\cdots[x_n \mapsto v_n]s_2)\ s)$$

すなわち $R_{S_2}(([x_1 \mapsto v_1]\cdots[x_n \mapsto v_n](\lambda x{:}S_1.\ s_2))\ s)$ である。s は任意に選ばれたので、$R_{S_1 \to S_2}$ の定義から

$$R_{S_1 \to S_2}([x_1 \mapsto v_1]\cdots[x_n \mapsto v_n](\lambda x{:}S_1.\ s_2))$$

がいえる。

T-APP の場合 $t = t_1\ t_2$

$x_1{:}T_1, \ldots, x_n{:}T_n \vdash t_1 : T_{11} \to T_{12}$

$x_1{:}T_1, \ldots, x_n{:}T_n \vdash t_2 : T_{11}$

$T = T_{12}$

帰納法の仮定より、$R_{T_{11} \to T_{12}}([x_1 \mapsto v_1]\cdots[x_n \mapsto v_n]t_1)$ かつ $R_{T_{11}}([x_1 \mapsto v_1]\cdots[x_n \mapsto v_n]t_2)$ である。$R_{T_{11} \to T_{12}}$ の定義から、

$$R_{T_{12}}(([x_1 \mapsto v_1]\cdots[x_n \mapsto v_n]t_1)\ ([x_1 \mapsto v_1]\cdots[x_n \mapsto v_n]t_2))$$

すなわち、$R_{T_{12}}([x_1 \mapsto v_1]\cdots[x_n \mapsto v_n](t_1\ t_2))$ である。 □

以上の帰結として正規化可能性が得られる。それには単に、補題 12.1.5 において t を閉じた項とし、すべての T について、すべての R_T の要素が正規化されることを思い出せばよい。

定理 12.1.6. [正規化] $\vdash t : T$ ならば、t は正規化可能である。 □

[証明] 補題 12.1.5 から、$R_T(t)$ である。ゆえに、t が正規化可能であることが補題 12.1.3 よりいえる。 □

演習 12.1.7. [推奨, ★★★] 本章の証明を拡張し、単純型付きラムダ計算がブール値 (図 8-1) と直積 (図 11-5) で拡張されても正規化できることを示せ。 □

12.2 注記

理論的な文献において、正規化可能性は、完全（非決定的）ベータ簡約を持つ計算体系の**強正規化**として定式化されるのが最も一般的である。標準的証明法は Tait [452] によって考案され、Girard によって System F（第 23 章を参照）の場合に一般化され（Girard [177]; Girard, Lafont, and Taylor [179]）、後に Tait [453] によって単純化された。本章で用いた説明法は、Martin Hofmann（私信）に従い、Tait の手法を値呼びに適応させたものである。論理関係という証明法の古典的な参考文献としては、Howard [223]、Tait [452]、Friedman [164]、Plotkin [384; 386]、Statman [441; 442; 443] がある。その他、意味論を扱う多くのテキストでも解説されている。例えば、Mitchell [322] や Gunter [188] が挙げられる。

Tait による正規化の証明は単純型付き項の評価アルゴリズムと全く同一のもので、**評価による正規化**あるいは**型主導部分評価**（Berger [44]; Danvy [130]）として知られる。Berger and Schwichtenberg [45] や Filinski [148; 149]、Reynolds [419] も参考にするとよい。

第 13 章

参照

本章では、Unit と参照を加えた単純型付きラムダ計算（図 13-1）を扱う。対応する OCaml 実装は fullref である。

本書ではここまで関数抽象、数やブール値といった基本的な型、レコードやバリアントといった構造化された型など、各種の**純粋な**言語機能を扱ってきた。これらの機能は、Haskell のような純粋関数型言語、ML のような「ほぼ関数型の」言語、C のような命令型言語、Java のようなオブジェクト指向言語、などほとんどのプログラミング言語の基幹をなしている。

ほとんどの実用的なプログラミング言語は、**純粋でない**機能も多数含んでいる。そのような機能はここまでの単純な意味論的枠組みでは説明ができない。特に、これらの言語における項の評価は、値を生成するだけではなく、書き換え可能な変数（参照セル、配列、書き換え可能なレコードのフィールドなど）への破壊的代入、ファイル・画面・ネットワーク接続への入出力、例外・ジャンプ・継続による制御の非局所的な移動、プロセス間の同期や通信への参加などを行うことがある。プログラミング言語の文献においては、このような計算の「副作用」は、より一般に**計算的作用**と呼ばれている。

本章では、計算的作用の一種である書き換え可能な参照を、これまで扱ってきた計算体系に加える方法を見てゆく。主な拡張は、**ストア**（あるいは**ヒープ**）を明示的に扱うようにすることである。この拡張を定義することは単純であるが、重要なのは、型保存定理（定理 13.5.3）の言明にどのような修正が必要になるかということである。この他の作用としては、制御の非局所的な移動である例外を、第 14 章で扱う。

13.1 導入

ほぼすべてのプログラミング言語[†1]は、事前に確保済みの記憶域の中身を変更する**破壊的代入**操作を、何らかの形で提供している。幾つかの言語、特に ML とその方言では、名前の束縛に用いる機構と、破壊的代入に用いる機構が分けられている。例えば、数 5 を値とする変数 x もあれば、5 が現在の中身である、書き換え可能なセルへの**参照**（または**ポインタ**）を値とする変数 y もあり、それらの違いがプログラマから見えるようになっている。x を他の数に足すことはできても、x への破壊的代入はできない。y が指すセルへの新しい値の破壊的代入には直接 y を使える（y:=84 と書く）が、y を plus の引数として直接使うことはできない。代わりに、現在の中身を得るため、明示的にその**参照先の値を取り出す**必要がある。この操作を !y と書く。他の多くの言語、特に、Java を含む C 系のすべての言語では、すべての変数名が書き換え可能なセルを参照する。また、現在の中身を得るために変数の参照先の値を取り出す操作は暗黙に行われる[†2]。

形式的な考察のためには、これらの機構を分けておくほうが便利である[†3]。本章では、ほぼ ML の

[†1] Haskell のような「純粋関数型」の言語でさえも、**モナド**のような拡張を通して、破壊的代入をサポートしている。

[†2] 厳密に言えば、C や Java での、型 T の変数のほとんどは、実際には型 Option(T) の値を持つセルへのポインタとして考えるべきである。これは変数の中身が、適切な値か特殊な値 null かのどちらかを取るという事実を反映している。

[†3] このように分離することは、言語設計の観点からも望ましいという有力な議論がある。セルをデフォルトで書き換え可能とするのではなく、意識した上で使わせることは、参照をあまり用いない、ほぼ関数的なプログラミングスタイルを促進する。このスタイルにより、プログラムの記述、メンテナンス、およびプログラムについての証明は著しく容易になる場合が多い。特に言語に並行性のような機能がある場合には有効である。

モデルに従って理論を構築する。C 風の言語に対してここでの内容を適用するには、単純に幾つかの区別をやめ、参照先の値の取り出しなど特定の操作を明示的ではなく暗黙に行えばよい。

基本的な事柄

参照への基本的な操作は、**確保**、**値の取り出し（参照先の）**、**破壊的代入**である。参照を確保するには、ref 演算子に新しいセルへの初期値を与えて用いる。

```
  r = ref 5;
▸ r : Ref Nat
```

型検査器のこの出力は、r の値がセルへの参照であり、またそのセルには常に数が格納されることを意味している。このセルの現在の値を読み取るには、参照先の値を取り出す演算子 ! を用いる。

```
  !r;
▸ 5 : Nat
```

セルに保存された値を変えるには、破壊的代入演算子を用いる。

```
  r := 7;
▸ unit : Unit
```

(破壊的代入の結果は、自明な値 unit である。11.2 節を見よ。) 再び r の参照先の値を取り出すと、更新後の値を確認できる。

```
  !r;
▸ 7 : Nat
```

副作用と逐次実行

破壊的代入の式が自明な値 unit を返すという事実は、11.3 節で定義した**逐次実行**記法と相性がよく、次のような書き方が可能になる。

```
  (r:=succ(!r); !r);
▸ 8 : Nat
```

これは次の複雑な書き方と等価である。

```
  (λ_:Unit. !r) (r := succ(!r));
▸ 9 : Nat
```

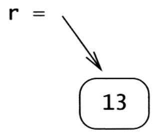
どちらも二つの式を順番に評価し、二つ目の式の値を返す。一つ目の式の型を Unit に限ることで、一つ目の値は、本当に自明な値である場合に限って捨てられるようになり、型検査器が一部の馬鹿なバグを見つける手助けになる。

二つ目の式もまた破壊的代入である場合には、逐次実行全体の型が Unit になるので、これを ; の左に置くことができ、さらに長い、破壊的代入の逐次実行の列を作れる。

```
  (r:=succ(!r); r:=succ(!r); r:=succ(!r); r:=succ(!r); !r);
▸ 13 : Nat
```

参照と別名付け

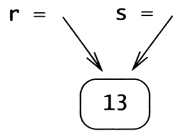
r に束縛された**参照**と、その参照が指すストアの中の**セル**との区別を意識しておくことは重要である。



r の値を別の変数 s に束縛するなどの方法で、r の複製を作ると、

```
  s = r;
▶ s : Ref Nat
```

参照だけが複製され（図中の**矢印**）、セルは複製されない。



これは次のように確かめられる。まず新しい値を s が指すセルへ破壊的に代入し、

```
  s := 82;
▶ unit : Unit
```

次に r を使ってそれを読み出す。

```
  !r;
▶ 82 : Nat
```

この参照 r と s を、この同一のセルに対する**別名**であるという。

演習 13.1.1. [★]　式 a = {ref 0, ref 0} と b = (λx:Ref Nat. {x,x}) (ref 0) とを評価したときの作用について、同様の図を描け。 □

共有状態

別名付けが存在する可能性がある場合、参照を含むプログラムについての証明はひどく扱いにくいものとなる。例えば、式 (r:=1; r:=!s) は、r に 1 を破壊的に代入し、すぐにそれを s が指すセルの現在の中身で上書きする。これは単一の破壊的代入 r:=!s と全く同じ作用を持つようだが、r と s が同一のセルに対する別名である文脈においてはその限りで**ない**。

一方で、参照の有用性をもたらす大きな要因が別名付けであることも確かである。特に、一つのプログラムの異なる部分の間で、「暗黙の通信チャネル」としての共有状態を作ることが、別名付けによって可能になる。例として、一つの参照セルと、その中身を操作する二つの関数を定義することを考える。

```
    c = ref 0;
  ▶ c : Ref Nat

    incc = λx:Unit. (c := succ (!c); !c);
  ▶ incc : Unit → Nat

    decc = λx:Unit. (c := pred (!c); !c);
  ▶ decc : Unit → Nat
```

incc を呼ぶと、

```
    incc unit;
  ▶ 1 : Nat
```

c が変更される。decc を呼ぶとそれがわかる。

```
    decc unit;
  ▶ 0 : Nat
```

incc と decc をレコードにまとめると、

```
    o = {i = incc, d = decc};
  ▶ o : {i:Unit→Nat, d:Unit→Nat}
```

この構造全体を一単位として受け渡せる。また、その要素を使って、共有状態 c に対する 1 の加算と 1 の減算の操作を実行できるようになる。これは実質的に、単純な形の**オブジェクト**を構成したことになる。第 18 章ではこのアイディアをさらに詳しく発展させる。

複合的な型への参照

参照セルは数だけを保持するようなものではない。上記のプリミティブな操作を使えば、関数型を含むどのような型の値についても参照を作れる。例えば、関数への参照を使って、数の配列を（あまり効率的ではないが）実装できる。NatArray と書いて、型 Ref (Nat→Nat) を意味することにする。

```
    NatArray = Ref (Nat→Nat);
```

新しい配列を作るには、参照セルを確保し、与えられた添字に対して常に 0 を返すような関数を入れる。

```
    newarray = λ_:Unit. ref (λn:Nat.0);
  ▶ newarray : Unit → NatArray
```

配列の要素を読むためには、単純にその関数を、望む添字に適用すればよい。

```
    lookup = λa:NatArray. λn:Nat. (!a) n;
  ▶ lookup : NatArray → Nat → Nat
```

この表現の面白いところは update 関数である。update は配列、添字と、その添字の場所に格納されるべき新しい値を受け取り、新しい関数を作る（その後、それを参照に格納する）ことで機能する。この新しい関数は、update に渡された添字に対しては、update に渡された新しい値を返し、他のすべての添字に対しては、それまで参照に格納されていた関数に読み出しを任せる。

```
    update = λa:NatArray. λm:Nat. λv:Nat.
                 let oldf = !a in
                 a := (λn:Nat. if equal m n then v else oldf n);
  ▶ update : NatArray → Nat → Nat → Unit
```

演習 13.1.2. [★★] updateをより小さく次のように定義したとしても、同じ動作をするか。

```
update = λa:NatArray. λm:Nat. λv:Nat.
           a := (λn:Nat. if equal m n then v else (!a) n);
```

□

参照を含む値への参照も場合によって非常に有用であり、書き換え可能なリストや木などのデータ構造を定義できる。(これらの構造は一般に、第 20 章で扱う**再帰型**をも含む。)

ガベージコレクション

参照を形式化する前準備として最後に残っているのが、記憶域の**解放**である。不要になった参照セルを解放するためのプリミティブな操作を、ここまでの議論では提供してこなかった。その代わり、多くの (ML と Java を含む) 現代的な言語と同じく、実行時システムが**ガベージコレクション**を行うことを当てにしている。これにより、プログラムから到達できなくなったセルは集められ、再利用される。これは単に言語設計の好みの問題**ではない**。明示的な解放の操作の存在下では、型安全性の達成が極めて困難になる。理由はよく知られた、**ぶら下がり参照**の問題である。例えば、数を保持するセルを確保し、そこへの参照を何らかのデータ構造に保存し、しばらくこれを利用し、そしてこれを解放した後、ブール値を保持する新しいセルを確保する。この新しいセルは同じ記憶域を再利用する可能性がある。すなわち、同一の記憶域セルに対して二つの名前がありうるということになる。一つは型 Ref Nat を持ち、もう一つは型 Ref Bool を持つ。

演習 13.1.3. [★★] これがどのように型安全性を侵害しうるのか示せ。 □

13.2 型付け

ref、:=、! についての型付け規則は、これらについて与えてきた振る舞いから自然に得られる。

$$\frac{\Gamma \vdash \mathtt{t}_1 : \mathtt{T}_1}{\Gamma \vdash \mathtt{ref}\ \mathtt{t}_1 : \mathtt{Ref}\ \mathtt{T}_1} \quad \text{(T-REF)}$$

$$\frac{\Gamma \vdash \mathtt{t}_1 : \mathtt{Ref}\ \mathtt{T}_1}{\Gamma \vdash \mathtt{!t}_1 : \mathtt{T}_1} \quad \text{(T-DEREF)}$$

$$\frac{\Gamma \vdash \mathtt{t}_1 : \mathtt{Ref}\ \mathtt{T}_1 \qquad \Gamma \vdash \mathtt{t}_2 : \mathtt{T}_1}{\Gamma \vdash \mathtt{t}_1\mathtt{:=}\mathtt{t}_2 : \mathtt{Unit}} \quad \text{(T-ASSIGN)}$$

13.3 評価

参照の扱いについてのより精妙な部分は、操作的振る舞いを形式化しようとしたときに現れる。その理由は、「Ref T の**値**は何であるべきか」との問いを考えると理解できる。演算子 ref を評価すると、何かが**行われる**べきである、という観察が重要である。すなわち、記憶域の確保がなされ、演算の返り値はこの記憶域への参照となる。

それでは、参照とは何か。

ほとんどのプログラミング言語の実装において、実行時ストアは実質的に、ただの巨大なバイト配列である。実行時システムは、この配列のどの部分が現在使われているかを常に追跡する。参照

セルを確保する必要が生じると、十分な大きさの領域がストアの未使用領域から確保され（整数セルには 4 バイト、`Float` を保存するセルには 8 バイト、など)、そこに使用中の印が付けられ、確保された領域の始点の添字（普通、32 または 64 ビット整数）が返される。この添字が参照である。

現在の目的のためには、そこまで具体的な扱いは必要ない。ここではストアを、バイト配列ではなく値の配列と考え、値ごとに異なる実行時表現の大きさを捨象できる。さらに参照（すなわち、この配列の添字）が数であるという事実も捨象できる。非解釈の集合 $\mathcal{L}$ を**ストアでの位置**の集合とし、参照を単にこの集合の要素とする。そして、ストアを位置 l から値への部分関数とする。ストアの上を動くメタ変数として、μ を用いる。参照は位置、すなわちストアの中を指す抽象化された添字である。ここではこの抽象的な性質を強調するため、**参照**や**ポインタ**に替えて、**位置**という語を使っていくことにする[†4]。

次に、これまでの操作的意味論に対して、ストアを考慮するよう拡張を行う必要がある。式の評価結果が一般に、評価時点でのストアの内容に依存することになるので、評価規則は引数として、項だけではなくストアをも取るようにする。さらに、項の評価がストアに対する副作用を引き起こし、その後の他の項の評価に影響を与えうるので、評価規則は新しいストアを返す必要がある。したがって、単一ステップの評価関係は、$\mathtt{t} \longrightarrow \mathtt{t}'$ から $\mathtt{t} \mid \mu \longrightarrow \mathtt{t}' \mid \mu'$ へと形を変えることになる。μ と μ' は始めと終わりのストアの状態である。これは実質的に、これまでの**抽象機械**の概念を拡張して、機械の状態が、単なるプログラムカウンタ（項として表現されている）だけではなく、プログラムカウンタに現在のストアの内容を合わせたものであるとしたことになる。

この変更を遂行するため、まず最初に既にあるすべての評価規則にストアを書き加える必要がある。

$$(\lambda \mathtt{x}{:}\mathtt{T}_{11}.\mathtt{t}_{12})\ \mathtt{v}_2 \mid \mu \longrightarrow [\mathtt{x} \mapsto \mathtt{v}_2]\mathtt{t}_{12} \mid \mu \qquad \text{(E-AppAbs)}$$

$$\frac{\mathtt{t}_1 \mid \mu \longrightarrow \mathtt{t}_1' \mid \mu'}{\mathtt{t}_1\ \mathtt{t}_2 \mid \mu \longrightarrow \mathtt{t}_1'\ \mathtt{t}_2 \mid \mu'} \qquad \text{(E-App1)}$$

$$\frac{\mathtt{t}_2 \mid \mu \longrightarrow \mathtt{t}_2' \mid \mu'}{\mathtt{v}_1\ \mathtt{t}_2 \mid \mu \longrightarrow \mathtt{v}_1\ \mathtt{t}_2' \mid \mu'} \qquad \text{(E-App2)}$$

最初の規則がストア μ を変更せずに返すことに注意されたい。関数適用は、それ自体は何の副作用も起こさない。他の二つの規則は単純に、前提から結論へ副作用を伝播させる。

次に項の**文法**に少し追加を行う。`ref` 式の評価結果はフレッシュな位置である。よって位置を、評価の結果となるようなものの集合、すなわち値の集合に含める必要がある。

v ::=		値：
	λx:T.t	ラムダ抽象値
	unit	unit 値
	l	ストアでの位置

値はすべて項でもあるので、項の集合も位置を含む必要があることになる。

[†4] このように位置を抽象的に扱うことで、C などの低水準言語に見られる**ポインタ算術**をモデル化することができなくなる。この制限は意図的なものである。ポインタ算術は時に（特にガベージコレクタなど実行時システムの低水準な部品を実装する場合）非常に有用であるものの、ほとんどの型システムでは追うことができない。例えばストアにおいて、位置 n が `Float` を保持しているという事実は、位置 $n+4$ の型について有用な情報を何も与えない。C において、ポインタ算術は型安全性侵害の元凶として悪名高い。

$\mathtt{t}$ ::=		項：
	$\mathtt{x}$	変数
	$\lambda\mathtt{x}{:}\mathtt{T}.\mathtt{t}$	ラムダ抽象
	$\mathtt{t}\ \mathtt{t}$	関数適用
	$\mathtt{unit}$	定数 unit
	$\mathtt{ref}\ \mathtt{t}$	参照の生成
	$!\mathtt{t}$	参照先の値の取り出し
	$\mathtt{t}{:=}\mathtt{t}$	破壊的代入
	l	ストアでの位置

もちろん、このように文法を拡張したからといって、**プログラマ**に、具体的な位置を明示的に含む項を書かせたいわけではない。そのような項は評価の中間結果としてのみ現れることになる。実際、本章での項の言語は**中間言語**を形式化したものと考えるべきであり、その機能をプログラマが直接利用できるものではない。

この拡張された文法を使って、位置とストアを扱う新しい構文についての評価規則を述べることができる。最初に述べるのは、参照先の値を取り出す式 $!\mathtt{t}_1$ の評価である。これにはまず $\mathtt{t}_1$ を、値になるまで簡約しなければならない。

$$\frac{\mathtt{t}_1 \mid \mu \longrightarrow \mathtt{t}_1' \mid \mu'}{!\mathtt{t}_1 \mid \mu \longrightarrow\ !\mathtt{t}_1' \mid \mu'} \qquad \text{(E-DEREF)}$$

$\mathtt{t}_1$ の簡約が終わると、残った式は、l を位置として $!l$ の形をしているはずである。関数や unit など他の種類の値を参照とみなして、参照先の値を取り出そうとするような項は、正しくない項である。この場合、簡約規則はそのまま行き詰まり状態になる。13.5 節で述べる型安全性についての諸性質によって、正しく型付けされた項が、決してこのような間違いをしないと保証される。

$$\frac{\mu(l) = \mathtt{v}}{!l \mid \mu \longrightarrow \mathtt{v} \mid \mu} \qquad \text{(E-DEREFLOC)}$$

次に述べるのは、破壊的代入の式 $\mathtt{t}_1{:=}\mathtt{t}_2$ の評価である。これにはまず $\mathtt{t}_1$ を、値（すなわち、位置）になるまで簡約しなければならない。

$$\frac{\mathtt{t}_1 \mid \mu \longrightarrow \mathtt{t}_1' \mid \mu'}{\mathtt{t}_1{:=}\mathtt{t}_2 \mid \mu \longrightarrow \mathtt{t}_1'{:=}\mathtt{t}_2 \mid \mu'} \qquad \text{(E-ASSIGN1)}$$

次に $\mathtt{t}_2$ を、値（こちらは任意の種類）になるまで簡約しなければならない。

$$\frac{\mathtt{t}_2 \mid \mu \longrightarrow \mathtt{t}_2' \mid \mu'}{\mathtt{v}_1{:=}\mathtt{t}_2 \mid \mu \longrightarrow \mathtt{v}_1{:=}\mathtt{t}_2' \mid \mu'} \qquad \text{(E-ASSIGN2)}$$

$\mathtt{t}_1$ と $\mathtt{t}_2$ の簡約が終わると、残った式は、$l{:=}\mathtt{v}_2$ の形をしているはずである。位置 l が $\mathtt{v}_2$ を持つようにストアを更新することで、これを実行する。

$$l{:=}\mathtt{v}_2 \mid \mu \longrightarrow \mathtt{unit} \mid [l \mapsto \mathtt{v}_2]\mu \qquad \text{(E-ASSIGN)}$$

（ここでの記法 $[l \mapsto \mathtt{v}_2]\mu$ は、「l を $\mathtt{v}_2$ に写し、他の位置については μ と同じように写すストア」を意味する。この評価ステップの返り値となる項が単に unit であることに注意されたい。興味がある結果は更新されたストアのほうである。）

最後に述べるのは、$\mathtt{ref}\ \mathtt{t}_1$ の形の式の評価である。これにはまず $\mathtt{t}_1$ を、値になるまで簡約しな

ければならない。

$$\frac{\mathtt{t}_1 \mid \mu \longrightarrow \mathtt{t}_1' \mid \mu'}{\mathtt{ref}\ \mathtt{t}_1 \mid \mu \longrightarrow \mathtt{ref}\ \mathtt{t}_1' \mid \mu'} \quad \text{(E-Ref)}$$

次に ref 自身を評価する。このためにはフレッシュな位置 l（すなわち、μ の定義域にいまだ含まれていないような位置）を一つ選び、新しい束縛 $l \mapsto \mathtt{v}_1$ によって μ を拡張した、新しいストアを生成する。

$$\frac{l \notin dom(\mu)}{\mathtt{ref}\ \mathtt{v}_1 \mid \mu \longrightarrow l \mid (\mu, l \mapsto \mathtt{v}_1)} \quad \text{(E-RefV)}$$

このステップの返り値となる項は、新たに確保された位置の名前 l である。

これらの評価規則はガベージコレクションを一切行わないことに注意されたい。ここでは単純に、評価が進むに従ってストアは無制限に大きくなってよいものとしている。これは評価結果の正しさには影響しないが（結局のところ「ガベージ（ごみ）」の定義は、ストアのうち到達できなくなった部分というだけであり、したがってそれ以降の評価には関与しない）、ここでの評価器を単純に実装すると、より洗練された評価器ならば中身がごみになった位置を再利用することで評価を続けられる場面でメモリ不足に陥ることもあるだろう。

演習 13.3.1. [★★★] ガベージコレクションをモデル化するためには、ここでの評価規則をどのように改良すればよいだろうか。またそのとき、加えた改良が正しいことを主張するために、どのような定理が証明されなければならないか。 □

13.4 ストア型付け

ここまでで参照の存在を許すように文法と評価規則を拡張し終えた。残るは新しい構文についての型付け規則を書き下すことと、当然必要な議論として、その健全性を確かめることである。必然的に、「位置の型は何か」が重要な問いとなる。

具体的な位置を含む項を評価する場合、結果の型は評価を始めたときのストアの中身に依存する。例えば、項 $!l_2$ をストア $(l_1 \mapsto \mathtt{unit}, l_2 \mapsto \mathtt{unit})$ で評価すると、結果は unit である。同じ項をストア $(l_1 \mapsto \mathtt{unit}, l_2 \mapsto \lambda\mathtt{x:Unit.x})$ で評価すると、結果は λx:Unit.x である。一つ目のストアについては、位置 l_2 は型 Unit を持ち、二つ目については、型 Unit→Unit を持つ。この観察からひとまず、位置の型付け規則を次のようにしてみる。

$$\frac{\Gamma \vdash \mu(l) : \mathtt{T}_1}{\Gamma \vdash l : \mathtt{Ref}\ \mathtt{T}_1}$$

つまり、位置 l の型を得るために、l が指す現在のストアの内容を参照し、その中身の型 $\mathtt{T}_1$ を計算する。すると位置の型は $\mathtt{Ref}\ \mathtt{T}_1$ となる。

こうして導入した規則を不整合のない状態にするには、もう少し構成を進める必要がある。項の型がストアに依存する形になったことで、型付け関係は事実上、三項関係（文脈、項、型）から四項関係（文脈、**ストア**、項、型）へと変わることになる。ストアは直観的には項の型を計算するのに使われる文脈の一部であるので、ストアをターンスタイル（⊢）の左に置き、この四項関係を $\Gamma \mid \mu \vdash \mathtt{t} : \mathtt{T}$ と書くことにしよう。これにより、参照の型付け規則は次の形となる。

$$\frac{\Gamma \mid \mu \vdash \mu(l) : \mathsf{T}_1}{\Gamma \mid \mu \vdash l : \mathsf{Ref}\ \mathsf{T}_1}$$

他の型付け規則もすべて、同様にストアを加えて拡張する。他の規則はストアについて特にすることはなく、単に前提から結論へストアを渡す。

しかしこの規則には二つの問題がある。一つ目の問題は、型検査がやや非効率的であることである。これは位置 l の型の計算が、l の現在の中身 v の型の計算を含んでいることによる。l が項 t に多数出現する場合、t の型付け導出の構築過程において、v の型も多数回再計算される。さらには、もし v 自身が位置を含む場合、**その位置**についても、出現のたびに型の再計算が必要になる。例えば、ストアが次の内容を含む場合、

$(l_1 \mapsto$ λx:Nat. 999,
 $l_2 \mapsto$ λx:Nat. (!l_1) x,
 $l_3 \mapsto$ λx:Nat. (!l_2) x,
 $l_4 \mapsto$ λx:Nat. (!l_3) x,
 $l_5 \mapsto$ λx:Nat. (!l_4) x)

l_5 の型の計算は、l_4、l_3、l_2、l_1 の型の計算を含むことになる。

二つ目の問題は、ここで試みた型付け規則はストアが**循環**を含む場合に何も導出できない可能性があることである。例えば、次のストアにおいては、位置 l_2 に有限の型付け導出が存在しない。

$(l_1 \mapsto$ λx:Nat. (!l_2) x,
 $l_2 \mapsto$ λx:Nat. (!l_1) x)

l_2 の型の計算には l_1 の型が必要であるが、その計算には l_2 が必要、と続くからである。循環参照構造は実用上現れるものであり（例えば、双方向リストを作るのに使われる）、ここでの型システムでも扱えるものであってほしい。

演習 13.4.1. [★] 具体的な項で、評価によってこの例の循環的なストアを作るものはあるか。 □

どちらの問題も、位置についての先ほどの型付け規則が、項における位置の出現ごとに位置の型の再計算を必要とすることが原因である。しかしこれは、直観的には不要のはずである。どのみち、位置が最初に生成されるときに、格納する初期値の型はわかっている。さらに、その位置に後で他の値を格納するとしても、それら他の値は、常に初期値と同じ型を持つはずである。つまり、ストアの位置ごとに、単一の型をはっきりと念頭に置いている。この型は位置が確保される時点で確定する。これらのあるべき型を集めて、全体として**ストア型付け**とすることができる。これはすなわち、位置を型へ写す有限の関数である。このような関数の上を動くメタ変数として、Σ を使う。

項 t がストア μ で評価されるとし、μ についてのストア型付け Σ が**与えられた**としよう。すると t の結果の型は Σ を使って計算できる。μ を直接見る必要はない。例えば、Σ が ($l_1 \mapsto$ Unit, $l_2 \mapsto$ Unit→Unit) であるとすると、!l_2 が型 Unit→Unit を持つと即座に推論できる。より一般に、位置の型付け規則は、ストア型付けを用いて次のように形式化し直せる。

$$\frac{\Sigma(l) = \mathsf{T}_1}{\Gamma \mid \Sigma \vdash l : \mathsf{Ref}\ \mathsf{T}_1} \qquad \text{(T-LOC)}$$

型付けはやはり四項関係ではあるが、具体的なストアではなく、ストア**型付け**についてパラメータ化されている。残りの型付け規則も同様に、ストア型付けによって拡張される。

これらの型付け規則が正確に評価の結果を予測するためには、当然ながら、評価に使われる具体

的なストアが、型検査のために仮定するストア型付けと実際に合うことが必要である。この条件は、ここまでに見てきたすべての計算体系における、自由変数についての状況と正確に類似する。すなわち、$\Gamma \vdash \mathtt{t} : \mathtt{T}$ としたとき、t の自由変数の型は Γ に列挙されているが、この型に合う値によって自由変数を置き換えれば、型 T の閉じた項が得られるということを代入補題（補題 9.3.8）が保証し、さらにこの項が評価されて何らかの結果に至るならば、その評価結果はやはり型 T を持つということを型保存定理（定理 9.3.9）が保証する。13.5 節では、ストアとストア型付けについての、これに類似する直観が形式化される様子を見る。

最後に注意すべき点だが、プログラマが実際に書く項を型検査する目的ならば、どのようなストア型付けを用いるべきか、ややこしいことをして推測する必要はない。先に述べたように、具体的な位置の定数は、評価の中間結果である項にのみ現れ、プログラマが書く言語には現れない。したがって、プログラマ由来の項については、**空の**ストア型付けを使って、簡単に型検査ができる。評価が進んで新しい位置が生成されるときには、新しく確保されたセルに置かれる初期値の型を見ることで、ストア型付けをどのように拡張すべきか、常にわかる。この直観は後に型保存定理（定理 13.5.3）の言明において形式化される。

位置についての扱いを終えてしまえば、他の新しい構文形式に対する型付け規則は実に単純なものである。型 $\mathtt{T}_1$ の値に対する参照を作る場合、参照自体は型 $\mathtt{Ref}\ \mathtt{T}_1$ を持つ。

$$\frac{\Gamma \mid \Sigma \vdash \mathtt{t}_1 : \mathtt{T}_1}{\Gamma \mid \Sigma \vdash \mathtt{ref}\ \mathtt{t}_1 : \mathtt{Ref}\ \mathtt{T}_1} \qquad \text{(T-Ref)}$$

ここではストア型付けを拡張する必要がないことに注意が必要である。なぜならば新しい位置の**名前**は実行時まで決定されることがなく、一方で Σ は、既に確保された記憶域セルと、その型との関連だけを記録するものだからである。

逆に、$\mathtt{t}_1$ が型 $\mathtt{Ref}\ \mathtt{T}_{11}$ を持つ位置へと評価されるならば、$\mathtt{t}_1$ が参照する先の値を取り出すことで、型 $\mathtt{T}_{11}$ の値が生成されることが保証される。

$$\frac{\Gamma \mid \Sigma \vdash \mathtt{t}_1 : \mathtt{Ref}\ \mathtt{T}_{11}}{\Gamma \mid \Sigma \vdash\ !\mathtt{t}_1 : \mathtt{T}_{11}} \qquad \text{(T-Deref)}$$

最後に、$\mathtt{t}_1$ が型 $\mathtt{Ref}\ \mathtt{T}_{11}$ を持つセルを表し、$\mathtt{t}_2$ の型もまた $\mathtt{T}_{11}$ であるならば、$\mathtt{t}_2$ をこのセルに格納することができる。

$$\frac{\Gamma \mid \Sigma \vdash \mathtt{t}_1 : \mathtt{Ref}\ \mathtt{T}_{11} \qquad \Gamma \mid \Sigma \vdash \mathtt{t}_2 : \mathtt{T}_{11}}{\Gamma \mid \Sigma \vdash \mathtt{t}_1 \mathtt{:=} \mathtt{t}_2 : \mathtt{Unit}} \qquad \text{(T-Assign)}$$

図 13-1 に、参照を加えた単純型付きラムダ計算の、型付け規則（および便利のため、文法と評価規則も）を要約する。

→ Unit Ref　　　$\lambda_\to$ の Unit（図 9-1 と図 11-2）を用いた拡張

構文

t ::=		項：
	unit	定数 unit
	x	変数
	λx:T.t	ラムダ抽象
	t t	関数適用
	ref t	参照の生成
	!t	参照先の値の取り出し
	t:=t	破壊的代入
	l	ストアでの位置
v ::=		値：
	unit	定数 unit
	λx:T.t	ラムダ抽象値
	l	ストアでの位置
T ::=		型：
	Unit	Unit 型
	T→T	関数の型
	Ref T	参照セルの型
μ ::=		ストア：
	$\varnothing$	空のストア
	$\mu, l \mapsto v$	位置への束縛
Γ ::=		文脈：
	$\varnothing$	空の文脈
	Γ, x:T	項変数の束縛
Σ ::=		ストア型付け：
	$\varnothing$	空のストア型付け
	Σ, l:T	位置の型付け

評価　　$\boxed{t \mid \mu \longrightarrow t' \mid \mu'}$

$$\frac{t_1 \mid \mu \longrightarrow t_1' \mid \mu'}{t_1\ t_2 \mid \mu \longrightarrow t_1'\ t_2 \mid \mu'} \quad \text{(E-APP1)}$$

$$\frac{t_2 \mid \mu \longrightarrow t_2' \mid \mu'}{v_1\ t_2 \mid \mu \longrightarrow v_1\ t_2' \mid \mu'} \quad \text{(E-APP2)}$$

$$(\lambda x{:}T_{11}.t_{12})\ v_2 \mid \mu \longrightarrow [x \mapsto v_2]t_{12} \mid \mu \quad \text{(E-APPABS)}$$

$$\frac{l \notin dom(\mu)}{\text{ref } v_1 \mid \mu \longrightarrow l \mid (\mu, l \mapsto v_1)} \quad \text{(E-REFV)}$$

$$\frac{t_1 \mid \mu \longrightarrow t_1' \mid \mu'}{\text{ref } t_1 \mid \mu \longrightarrow \text{ref } t_1' \mid \mu'} \quad \text{(E-REF)}$$

$$\frac{\mu(l) = v}{!l \mid \mu \longrightarrow v \mid \mu} \quad \text{(E-DEREFLOC)}$$

$$\frac{t_1 \mid \mu \longrightarrow t_1' \mid \mu'}{!t_1 \mid \mu \longrightarrow !t_1' \mid \mu'} \quad \text{(E-DEREF)}$$

$$l{:=}v_2 \mid \mu \longrightarrow \text{unit} \mid [l \mapsto v_2]\mu \quad \text{(E-ASSIGN)}$$

$$\frac{t_1 \mid \mu \longrightarrow t_1' \mid \mu'}{t_1{:=}t_2 \mid \mu \longrightarrow t_1'{:=}t_2 \mid \mu'} \quad \text{(E-ASSIGN1)}$$

$$\frac{t_2 \mid \mu \longrightarrow t_2' \mid \mu'}{v_1{:=}t_2 \mid \mu \longrightarrow v_1{:=}t_2' \mid \mu'} \quad \text{(E-ASSIGN2)}$$

続く...

▲ 図 13-1.　参照

型付け $\boxed{\Gamma \mid \Sigma \vdash \mathtt{t} : \mathtt{T}}$

$$\Gamma \mid \Sigma \vdash \mathtt{unit} : \mathtt{Unit} \qquad \text{(T-UNIT)}$$

$$\frac{\mathtt{x{:}T} \in \Gamma}{\Gamma \mid \Sigma \vdash \mathtt{x} : \mathtt{T}} \qquad \text{(T-VAR)}$$

$$\frac{\Gamma, \mathtt{x{:}T_1} \mid \Sigma \vdash \mathtt{t_2} : \mathtt{T_2}}{\Gamma \mid \Sigma \vdash \lambda\mathtt{x{:}T_1.t_2} : \mathtt{T_1 \to T_2}} \qquad \text{(T-ABS)}$$

$$\frac{\Gamma \mid \Sigma \vdash \mathtt{t_1} : \mathtt{T_{11} \to T_{12}} \qquad \Gamma \mid \Sigma \vdash \mathtt{t_2} : \mathtt{T_{11}}}{\Gamma \mid \Sigma \vdash \mathtt{t_1\ t_2} : \mathtt{T_{12}}} \qquad \text{(T-APP)}$$

$$\frac{\Sigma(l) = \mathtt{T_1}}{\Gamma \mid \Sigma \vdash l : \mathtt{Ref\ T_1}} \qquad \text{(T-LOC)}$$

$$\frac{\Gamma \mid \Sigma \vdash \mathtt{t_1} : \mathtt{T_1}}{\Gamma \mid \Sigma \vdash \mathtt{ref\ t_1} : \mathtt{Ref\ T_1}} \qquad \text{(T-REF)}$$

$$\frac{\Gamma \mid \Sigma \vdash \mathtt{t_1} : \mathtt{Ref\ T_{11}}}{\Gamma \mid \Sigma \vdash \mathtt{!t_1} : \mathtt{T_{11}}} \qquad \text{(T-DEREF)}$$

$$\frac{\Gamma \mid \Sigma \vdash \mathtt{t_1} : \mathtt{Ref\ T_{11}} \qquad \Gamma \mid \Sigma \vdash \mathtt{t_2} : \mathtt{T_{11}}}{\Gamma \mid \Sigma \vdash \mathtt{t_1{:=}t_2} : \mathtt{Unit}} \qquad \text{(T-ASSIGN)}$$

▲ 図 13-1. 参照、続き

13.5 安全性

本章では最後に、標準的な型安全性についての性質が、参照を加えた計算体系についても成り立つことを確認する。進行定理（「正しく型付けされた項は行き詰まり状態にならない」）はこれまでとほぼ同じ言明で、証明も同様である（定理 13.5.7 を見よ）。つまり、新しい構文を扱うために、幾つかの単純な場合分けを証明に加えるだけで足りる。保存定理のほうはもう少し興味深いので、こちらを先に見ることにしよう。

ここまでに、評価関係と型付け関係の両方を拡張したので（前者については評価の始めと終わりのストアによって、後者についてはストア型付けによって）、保存の言明にこれらのパラメータを加える変更が必要となる。しかし両者の関係について何ら述べることなしに、ストアとストア型付けをただ付け加えることは、明らかに不可能である。

$\Gamma \mid \Sigma \vdash \mathtt{t} : \mathtt{T}$ かつ $\mathtt{t} \mid \mu \longrightarrow \mathtt{t'} \mid \mu'$ ならば、$\Gamma \mid \Sigma \vdash \mathtt{t'} : \mathtt{T}$ である。　　**（これは間違い！）**

ストアの中の値の型について何か仮定を幾つか置いて型検査を行い、次にその仮定を侵害するようなストアを使って評価を行えば、結果はでたらめなものになる。次の定義の条件部分は、必要な制約条件を述べている。

定義 13.5.1. ストア μ が、型付け文脈 Γ とストア型付け Σ とに関して**正しく型付けされている**とは、$dom(\mu) = dom(\Sigma)$、かつ、任意の $l \in dom(\mu)$ について $\Gamma \mid \Sigma \vdash \mu(l) : \Sigma(l)$ となることである。これを $\Gamma \mid \Sigma \vdash \mu$ と書く[†5]。 □

直観的には、ストア μ の中の任意の値が、ストア型付け Σ によって予測される型を持つならば、μ は Σ に対して整合的である。

演習 13.5.2. [★★]　$\Gamma \mid \Sigma_1 \vdash \mu$ と $\Gamma \mid \Sigma_2 \vdash \mu$ とを同時に満たすように、文脈 Γ、ストア μ、および相異なるストア型付け Σ_1 と Σ_2 を取れるか。 □

[†5] （訳注：原著正誤表より）本書で扱う範囲では、文脈は任意のものではなく、空の文脈でのみ考えればよい。

ここまでで、求めている保存の性質にある程度近づけた。

もし、

$\Gamma \mid \Sigma \vdash \mathsf{t} : \mathsf{T}$

$\Gamma \mid \Sigma \vdash \mu$

$\mathsf{t} \mid \mu \longrightarrow \mathsf{t}' \mid \mu'$

ならば、$\Gamma \mid \Sigma \vdash \mathsf{t}' : \mathsf{T}$ である。　　　**(前回ほどは間違っていない)**

この言明は、ほとんどすべての評価規則についてうまくいくが、確保の規則 E-RefV についてだけは問題がある。問題とは、この規則が評価始めのストアよりも大きな定義域を持つストアを生成することであり、これにより上の言明の結論部分が偽になってしまう。すなわち、もし μ' がフレッシュな位置 l の束縛を含むとすると、l は Σ の定義域にはないことになり、したがって Σ を使って t'（間違いなく l が出現している）を型付けできない。

評価が進むに従ってストアの大きさが増すのは明らかなので、ストア型付けについてもまた、拡大を許すようにしなければならない。これを考慮することで、最終的な（正しい）型保存の性質を述べられる。

定理 13.5.3.［保存］ もし、

$\Gamma \mid \Sigma \vdash \mathsf{t} : \mathsf{T}$

$\Gamma \mid \Sigma \vdash \mu$

$\mathsf{t} \mid \mu \longrightarrow \mathsf{t}' \mid \mu'$

ならば、ある $\Sigma' \supseteq \Sigma$ が存在して、

$\Gamma \mid \Sigma' \vdash \mathsf{t}' : \mathsf{T}$

$\Gamma \mid \Sigma' \vdash \mu'$

となる。 □

保存定理は、ストア型付け $\Sigma' \supseteq \Sigma$（すなわち、すべての古い位置が指す型について Σ と一致するもの）であって、新しい項 t' が Σ' に関して正しく型付けされるようなものについて、単にその**存在だけ**を主張していることに注意されたい。実際に Σ' がどのようなものであるかは述べられていない。もちろん直観的には、Σ' は Σ であるか、さもなくば正確に $(\Sigma, l \mapsto \mathsf{T}_1)$ であるとわかる。ここで、l は新たに確保された位置（μ' の定義域の新しい要素）であり、T_1 は拡張されたストア $(\mu, l \mapsto \mathsf{v}_1)$ において l に束縛された初期値の型である。しかし、Σ' を明示しても定理の言明が煩雑になるだけである。上で述べた弱い言明は既に、評価ステップの任意の**列**が正しい型付けを保存することを繰り返し「クランクを回す」ことで結論付けるには正しい形である（結論が仮定を含意しているので）。これを進行の性質と組み合わせることで、いつもの通り「正しく型付けされたプログラムはおかしくなることはない」ということを保証できる。

保存を証明するために、幾つかの技術的な補題が必要となる。一つ目は、標準的な代入補題（補題 9.3.8）の簡単な拡張である。

補題 13.5.4.［代入］ もし、$\Gamma, \mathsf{x}{:}\mathsf{S} \mid \Sigma \vdash \mathsf{t} : \mathsf{T}$ かつ $\Gamma \mid \Sigma \vdash \mathsf{s} : \mathsf{S}$ ならば、$\Gamma \mid \Sigma \vdash [\mathsf{x} \mapsto \mathsf{s}]\mathsf{t} : \mathsf{T}$ である。 □

［証明］ 補題 9.3.8 と同様。 □

次の補題は、ストアにおけるセルの中身を、適当な型を持つ新しい値で置き換えても、ストア全

体の型は変わらないということを主張している。

補題 13.5.5. もし、

$$\Gamma \mid \Sigma \vdash \mu$$
$$\Sigma(l) = \mathtt{T}$$
$$\Gamma \mid \Sigma \vdash \mathtt{v} : \mathtt{T}$$

ならば、$\Gamma \mid \Sigma \vdash [l \mapsto \mathtt{v}]\mu$ である。 □

[証明] $\Gamma \mid \Sigma \vdash \mu$ の定義より直ちに成り立つ。 □

最後に、ストアに対する弱化補題のようなものが必要になる。これは、もしストアが新しい位置を加えて拡張された場合、拡張されたストアにおいても、元のストアで型付けされていた項には同じ型を付けられることを主張するものである。

補題 13.5.6. もし、$\Gamma \mid \Sigma \vdash \mathtt{t} : \mathtt{T}$ かつ $\Sigma' \supseteq \Sigma$ ならば、$\Gamma \mid \Sigma' \vdash \mathtt{t} : \mathtt{T}$ である。 □

[証明] 簡単な帰納法による。 □

ここまでの準備をもって主たる保存定理を証明できる。

[定理 13.5.3 の証明] 評価導出についての単純な帰納法による。上記の補題と、型付け規則の逆転の性質（補題 9.3.1 の単純な拡張）を用いる。 □

進行定理（定理 9.3.5）の言明もまた、ストアとストア型付けとを考慮して拡張する必要がある。

定理 13.5.7. [進行] $\mathtt{t}$ を閉じた、正しく型付けされた項（すなわち、何か $\mathtt{T}$ と Σ が存在して $\emptyset \mid \Sigma \vdash \mathtt{t} : \mathtt{T}$ となる）とする。このとき $\mathtt{t}$ は値であるか、さもなくば $\emptyset \mid \Sigma \vdash \mu$ となる任意のストア μ について、項 $\mathtt{t}'$ とストア μ' とが存在し $\mathtt{t} \mid \mu \longrightarrow \mathtt{t}' \mid \mu'$ となる。 □

[証明] 型付け導出に対する単純な帰納法による。定理 9.3.5 のパターンに従う。(標準形補題は、補題 9.3.4 に加えて、型 `Ref T` の値がすべて位置であるという主張と `Unit` についての同様の主張の二つの場合が必要になる。) □

演習 13.5.8. [推奨, ★★★] 本章の評価規則は、正しく型付けされた項に対して正規化可能性を持つか。持つならば証明せよ。持たないならば、正しく型付けされた階乗関数をこの計算体系で書け (数とブール値についての拡張をした上で)。 □

13.6 注記

本章での説明は、Harper [192; 193] を出典とする。同様の様式による説明は、Wright and Felleisen [500] でも行われている。

参照（および他の計算的作用）を ML 風の多相型推論と組み合わせることは、幾つかのかなり込み入った問題を引き起こし（22.7 節を見よ)、研究文献において多くの注意を惹きつけることとなった。Tofte [464]、Hoang et al. [211]、Jouvelot and Gifford [244]、Talpin and Jouvelot [454]、Leroy and Weis [284]、Wright [498]、Harper [192; 193]、および、これらからの参考文献を見よ。

別名付けの可能性についての静的予測は、コンパイラの実装（そこでは**別名解析**と呼ばれる）と、プログラミング言語理論との両方において、長きにわたる問題である。初期に影響の大きかった

Reynolds [410; 416] による試みは、**構文による干渉の制御**という言葉を生んだ。これらの考えは、近年新たに活性化している。O'Hearn et al. [347] と Smith et al. [435] とを見よ。別名付けについてのより一般的な証明技法は、Reynolds [412] と Ishtiaq and O'Hearn [234]、およびこれらからの参考文献において議論されている。

ガベージコレクションについての包括的な議論は、Jones and Lins [243] にある。より意味論的な扱いは Morrisett et al. [327] にある。

この作用の原因を探さねばなりません、
言い替えましょう、この不良の原因を探さねばなりません、
なんとなればこの不良な作用は、原因によって生じておりますゆえ。
—ハムレット 第 2 幕第 2 場 101 行目

月を指差すその指は、しかし月とは異なる。 —仏教徒の諺

第14章
例外

本章で扱う体系は、単純型付きラムダ計算（図 9-1）を、例外や例外処理のためのさまざまなプリミティブ（図 14-1 や図 14-2）で拡張したものである。最初の拡張の OCaml 実装が `fullerror` である。値を持つ例外を備えた言語（図 14-3）は実装されていない。

第 13 章では、純粋な単純型付きラムダ計算の単純な操作的意味論を、書き換え可能な参照で拡張する方法を見た。また、この拡張が型付け規則や型安全性の証明に与える影響を考えた。この章では例外の送出と処理という、元の計算モデルに対する別の拡張を考える。

現実世界のプログラミングには、関数が何らかの理由で処理を実行できないことを呼び出し元に伝えなければならない事例があふれている。例えば、ゼロ除算やオーバーフローが起きるような計算をした場合、検索キーが辞書から見つからなかった場合、配列の添字が範囲外だった場合、ファイルが見つからなかったり開けなかったりした場合、システムがメモリ不足になったりユーザがプロセスを殺したりなどの破滅的な事象が発生した場合などである。

このような例外的な状況の一部は、11.10 節で見たように、関数がバリアント（やオプション）を返す形で伝えることができる。しかしそのような例外的な状況が本当に**例外的**である場合は、関数の呼び出し元すべてにいちいち例外が起きる可能性を処理させたくないと思うかもしれない。それよりは、例外的な状況が起きた場合、プログラムのより高水準で定義された**例外ハンドラ**に制御を直接移したい、もしくは、実際（その例外的な状況が十分に稀であるか、呼び出し元ではどうにせよ回復のしようがない状況である場合には）プログラムを単に中断させたいと思うかもしれない。本章ではまず後者の場合、つまり例外が起きたらプログラム全体が中断する場合を考える（14.1 節）。それから例外を捕捉しそこから回復する仕組みを追加する（14.2 節）。最後に、それらの両方の仕組みを洗練し、例外発生地点とハンドラの間で受け渡される追加のデータをプログラマが指定できるようにする（14.3 節）。

14.1 例外の送出

単純型付きラムダ計算に、考えうる最も単純な例外通知の仕組みである、項 `error` を追加するところから始めよう。この項は評価されると、それが含まれる項の評価を完全に中断する。必要な拡張の詳細を図 14-1 に示す。

`error` の規則を書くにあたっては、我々の操作的意味論において「異常終了」をどのように形式化するかが設計上の主な決定となる。ここでは、異常終了したプログラムの結果を `error` 自身にするという単純な方法を用いる。規則 E-AppErr1 と E-AppErr2 がこの挙動を表している。E-AppErr1 は、適用の左側を値に簡約しようとしているときに項 `error` が現れた場合、その適用の結果として直ちに `error` を返す、という規則である。同様に E-AppErr2 は、適用の引数を値に簡約している最中に `error` が現れた場合、その適用の簡約はやめて直ちに `error` を返す、という規則である。

`error` を値の構文に含めず、項の構文だけに含めていることに注意されたい。これによって、E-AppAbs と E-AppErr2 の規則の左側が重ならないことを保証している。つまり、以下の項を評価するとき、

```
(λx:Nat.0) error
```

適用を行うか（結果として 0 を返す）、中断するかが曖昧にならず、後者しかありえないことが保証

→ エラー　　　　λ→（図 9-1）の拡張

新しい構文形式

t ::=	…	項：
	error	実行時エラー

新しい評価規則　　$t \longrightarrow t'$

$$\texttt{error}\ t_2 \longrightarrow \texttt{error} \qquad \text{(E-AppErr1)}$$

$$v_1\ \texttt{error} \longrightarrow \texttt{error} \qquad \text{(E-AppErr2)}$$

新しい型付け規則　　$\Gamma \vdash t : T$

$$\Gamma \vdash \texttt{error} : T \qquad \text{(T-Error)}$$

▲図 14-1.　エラー

される。同様に、E-AppErr2 において（任意の項の上を動く t_1 ではなく）メタ変数 v_1 を使っていることから、たとえ右側が error であっても、適用の左側が値に簡約されるまで、評価器はそれを中断できない。よって、以下のような項は、

```
(fix (λx:Nat.x)) error
```

中断ではなく発散する。以上の条件により、評価関係は依然として決定的になることが保証される。

型付け規則 T-Error も興味深い。任意の文脈で例外を送出したくなるかもしれないので、項 error はどんな型でも持つことが許されている。

```
(λx:Bool.x) error;
```

の場合は Bool 型を持つ。

```
(λx:Bool.x) (error true);
```

の場合は Bool→Bool 型を持つ。

この error の型の柔軟性は、型検査アルゴリズムを実装する際に幾つかの困難を引き起こす。言語中の型付け可能な任意の項は一意な型を持つという性質（定理 9.3.3）が壊れるからである。これはさまざまな方法で対処できる。部分型付けのある言語であれば、error に対して、最小の型 Bot を割り当てればよい（15.4 節）。Bot は、必要に応じて任意の他の型に**昇格**できる。パラメータ多相（第 23 章）を持つ言語であれば、error に多相型 ∀X.X を与えることができる（∀X.X は任意の型に具体化できる）。どちらのトリックも、error が持ちうる無限通りの型を一つの型でまとめて表現させるものである。

演習 14.1.1. [★]　単に、error が使われる文脈ごとに、プログラマが意図する型を注記するよう義務付けたほうが簡単ではないだろうか？ □

例外を持つ言語の型保存の性質はいつもと同じである。もし項が型 T を持ち、さらにその項の評価が 1 ステップ進むならば、その結果も依然として型 T を持つ。しかし進行の性質には少し改良が必要である。元々の形では、正しく型付けされたプログラムは、値に評価される（か発散する）、という性質だった。しかし今回、値ではない正規形 error を導入したので、それもまた、正しく型付けされたプログラムの評価結果になりうる。これを許すように進行を言い直す必要がある。

定理 14.1.2. **[進行]**　t は閉じていて、正しく型付けされた正規形とする。このとき、t は値であるか、または t = error である。 □

→ エラー 捕捉 | | λ→ とエラー（図 14-1）の拡張

新しい構文形式

t ::= ... 項：

try t with t エラーの捕捉

新しい評価規則 $\boxed{t \longrightarrow t'}$

$$\texttt{try } v_1 \texttt{ with } t_2 \longrightarrow v_1 \qquad \text{(E-TryV)}$$

$$\texttt{try error with } t_2 \longrightarrow t_2 \qquad \text{(E-TryError)}$$

$$\frac{t_1 \longrightarrow t_1'}{\texttt{try } t_1 \texttt{ with } t_2 \longrightarrow \texttt{try } t_1' \texttt{ with } t_2} \qquad \text{(E-Try)}$$

新しい型付け規則 $\boxed{\Gamma \vdash t : T}$

$$\frac{\Gamma \vdash t_1 : T \qquad \Gamma \vdash t_2 : T}{\Gamma \vdash \texttt{try } t_1 \texttt{ with } t_2 : T} \qquad \text{(T-Try)}$$

▲図 14-2. エラー処理

14.2 例外の処理

error の評価規則は「呼び出しスタックの巻き戻し」と考えられる。error がトップレベルに伝播するまで、残った関数呼び出しは捨てられる。これはまさに、例外のある言語の実際の実装で起きていることである。呼び出しスタックは、起動中の関数呼び出しごとの**起動記録**の集合からなる。例外を送出すると、呼び出しスタックが空になるまで起動記録が取り除かれる。

例外のあるほとんどの言語では、呼び出しスタックに**例外ハンドラ**を設置できる。例外が発生すると、例外ハンドラが現れるまで呼び出しスタックから起動記録が取り除かれ、そのハンドラから評価が続く。つまり、例外は制御の非局所的な移動として機能する。移動先は、最も新しく設置された例外ハンドラ（つまり呼び出しスタック上で最も近いもの）である。

図 14-2 に要約した我々の例外ハンドラの形式化は、ML と Java の両方に類似している。式 try t_1 with t_2 は、「t_1 の評価が中断しなければその結果を返し、中断した場合は代わりにハンドラ t_2 を評価する」ことを意味する。評価規則 E-TryV は、t_1 が値 v_1 に簡約された場合、try はもはや不要とわかるので捨ててよいということを表す。一方 E-TryError は、t_1 の評価が error になった場合、try を t_2 に置き換えてそこから評価を続けなければならないということを表す。E-Try は、t_1 が値か error に簡約されるまで、t_1 の簡約を続け、t_2 はそのままにしておかなければならないことを表す。

try の型付け規則は、その操作的意味論から直接得られる。try 全体の評価結果は、本体 t_1 の結果か、さもなくばハンドラ t_2 の結果のどちらかになる。これらが同じ型 T を持つよう、単純に要求すればよい。T は try の型にもなる。

型安全性とその証明は前節と本質的に変わらないままとなる。

14.3 値を持つ例外

14.1 節と 14.2 節で導入した仕組みでは、関数が呼び出し元に「何か異常が起きた」ことを伝えることしかできない。一般的には、「どのような異常が起きたか」という何らかの追加情報も送り返せると便利である。この情報によって、ハンドラが取るべき対応（回復して再実行するか、ユーザにわかりやすいエラーメッセージを出すか）が変わるかもしれないからである。

図 14-3 に、各例外（exception）が値を持つよう、例外処理の構成要素を拡張する方法を示す。この値の型を T_{exn} と書く。当面、この型の正確な性質は定めないでおく。後で幾つかの選択肢を議

→ 例外 　　　　λ→（図 9-1）の拡張

新しい構文形式

$\mathtt{t} ::= \ldots$ 　　　項：

$\mathtt{raise\ t}$ 　　　例外の送出

$\mathtt{try\ t\ with\ t}$ 　　　例外の処理

新しい評価規則 　　　$\boxed{\mathtt{t} \longrightarrow \mathtt{t'}}$

$$(\mathtt{raise}\ \mathtt{v}_{11})\ \mathtt{t}_2 \longrightarrow \mathtt{raise}\ \mathtt{v}_{11} \quad \text{(E-AppRaise1)}$$

$$\mathtt{v}_1\ (\mathtt{raise}\ \mathtt{v}_{21}) \longrightarrow \mathtt{raise}\ \mathtt{v}_{21} \quad \text{(E-AppRaise2)}$$

$$\frac{\mathtt{t}_1 \longrightarrow \mathtt{t}_1'}{\mathtt{raise}\ \mathtt{t}_1 \longrightarrow \mathtt{raise}\ \mathtt{t}_1'} \quad \text{(E-Raise)}$$

$$\mathtt{raise}\ (\mathtt{raise}\ \mathtt{v}_{11}) \longrightarrow \mathtt{raise}\ \mathtt{v}_{11} \quad \text{(E-RaiseRaise)}$$

$$\mathtt{try}\ \mathtt{v}_1\ \mathtt{with}\ \mathtt{t}_2 \longrightarrow \mathtt{v}_1 \quad \text{(E-TryV)}$$

$$\mathtt{try}\ \mathtt{raise}\ \mathtt{v}_{11}\ \mathtt{with}\ \mathtt{t}_2 \longrightarrow \mathtt{t}_2\ \mathtt{v}_{11} \quad \text{(E-TryRaise)}$$

$$\frac{\mathtt{t}_1 \longrightarrow \mathtt{t}_1'}{\mathtt{try}\ \mathtt{t}_1\ \mathtt{with}\ \mathtt{t}_2 \longrightarrow \mathtt{try}\ \mathtt{t}_1'\ \mathtt{with}\ \mathtt{t}_2} \quad \text{(E-Try)}$$

新しい型付け規則 　　　$\boxed{\Gamma \vdash \mathtt{t} : \mathtt{T}}$

$$\frac{\Gamma \vdash \mathtt{t}_1 : \mathtt{T}_{exn}}{\Gamma \vdash \mathtt{raise}\ \mathtt{t}_1 : \mathtt{T}} \quad \text{(T-Exn)}$$

$$\frac{\Gamma \vdash \mathtt{t}_1 : \mathtt{T} \qquad \Gamma \vdash \mathtt{t}_2 : \mathtt{T}_{exn} \to \mathtt{T}}{\Gamma \vdash \mathtt{try}\ \mathtt{t}_1\ \mathtt{with}\ \mathtt{t}_2 : \mathtt{T}} \quad \text{(T-Try)}$$

▲ 図 14-3. 値を持つ例外

論する。

原子項 error は、項構築子 raise t に置き換える。t は例外ハンドラに渡したい追加情報である。try の文法は変わらないが、try $\mathtt{t}_1$ with $\mathtt{t}_2$ の中のハンドラ $\mathtt{t}_2$ は、追加情報を引数として受け取る**関数**として解釈されるようになる。

評価規則 E-TryRaise はこの挙動を実装している。本体 $\mathtt{t}_1$ から raise によって運ばれる追加情報を受け取り、それをハンドラ $\mathtt{t}_2$ に渡す。E-AppRaise1 と E-AppRaise2 は、図 14-1 の E-AppErr1 や E-AppErr2 とちょうど同じように、適用を通過して例外を伝播させる。ただしこれらの規則は、追加情報が値である例外の伝播しか許していないことに注意されたい。追加情報自体が評価を必要とするような raise を評価しようとする場合は、これらの規則は適用できず、まず E-Raise を用いて追加情報を評価しなければならない。E-RaiseRaise は、ある例外と共に送られる追加情報を評価**している最中**に起きるかもしれない、別の例外を伝播させる。E-TryV は、14.2 節と同様に、本体が値に簡約されたら try が捨てられる、ということを表している。E-Try は、値か raise になるまで try の本体を評価させる。

型付け規則は、振る舞いに関するこれまでの変更を反映したものとなる。T-Exn では追加情報が型 $\mathtt{T}_{exn}$ を持つように要求する。このとき raise 全体には、文脈から要求されうる任意の型 T が与えられる。T-Try ではハンドラ $\mathtt{t}_2$ が型 $\mathtt{T}_{exn}$ の追加情報を受け取り、$\mathtt{t}_1$ と同じ型の結果を返す関数であることを検査する。

最後に、型 $\mathtt{T}_{exn}$ の選択肢を幾つか考えよう。

(1) $\mathtt{T}_{exn}$ を単に Nat とすることが考えられる。これは例えば Unix オペレーティングシステムの関数などで使われる errno の慣習に相当する。各システムコールが整数の「エラーコード」を返す。0 は成功を表し、それ以外の値はさまざまな例外状況を表す。

(2) $\mathtt{T}_{exn}$ を String とすることも考えられる。表からエラー番号を引く必要がなく、例外送出側が望めばより説明的なメッセージを構成できる。このような自由度を追加するデメリットは、

例外ハンドラは何が起きたか知るために、それらの文字列を**構文解析**する必要があるかもしれないことである。

(3) より情報量のある例外を渡す能力を維持しつつ文字列の構文解析を避けるには、T_{exn} を**バリアント型**として定義すればよい。

```
Texn  =  <divideByZero:     Unit,
          overflow:         Unit,
          fileNotFound:     String,
          fileNotReadable:  String,
          ...>
```

この仕組みならば、ハンドラは簡単な case 式で例外の種類を見分けられる。さらに、異なる例外は異なる型の追加情報を持てる。例えば、divideByZero のような例外は余分な手荷物を持つ必要はないが、fileNotFound はエラーが起きたときにどのファイルを開こうとしていたのか示す文字列を持てる。

この選択肢の問題点は、任意のプログラムにより送出されうる例外の集合（つまり、バリアント型 T_{exn} のタグの集合）を**事前に**完全に固定しなければならず、かなり柔軟性がないということである。これではプログラマがアプリケーション固有の例外を宣言する余地がない。

(4) 同じアイディアを洗練して、T_{exn} を**拡張可能バリアント型**にすることでユーザ定義例外を許すことができる。ML はこのアイディアを採用し、exn と呼ばれる単一の拡張可能バリアント型を提供している[†1]。ML の宣言 exception l of T は、この文脈では次のように解釈できる。「l が、既にバリアント型 T_{exn} に含まれるどのタグとも異なることを確認し[†2]、今後は T_{exn} を $<l_1{:}T_1 \ldots l_n{:}T_n, l{:}T>$ とする。ただし $l_1{:}T_1$ から $l_n{:}T_n$ は、この宣言の前からあったバリアントである。」

ML の例外送出の構文は raise l(t) である。l は現在のスコープで定義されている例外のタグである。これは、本章の単純な raise とタグ付け演算子との組み合わせと解釈できる。

$$\texttt{raise l(t)} \quad \stackrel{\text{def}}{=} \quad \texttt{raise (<l=t> as } T_{exn}\texttt{)}$$

同様に ML の try 構文は、本章の単純な try に case を加えた構文の糖衣とみなせる。

```
try t with l(x) → h   ≝   try t with
                            λe:Texn. case e of
                                        <l=x> ⇒ h
                                      | _ ⇒ raise e
```

この case は、送出された例外が l でタグ付けされているかどうかを検査する。もしされていれば、例外が運んでいる値を変数 x に束縛し、ハンドラ h を評価する。されていなければ、パターン _ にマッチし例外は**再送出**される。例外は、その例外を処理しようとしているハンドラに到達するまで（時に捕捉されては再送出されて）伝播し続ける。トップレベルに到達した場合は、プログラム全体が中断する。

(5) Java はユーザ定義例外をサポートするため、拡張可能バリアントの代わりにクラスを使っている。この言語は組み込みクラス Throwable を提供している。Throwable やその任意のサ

[†1] さらに進んで、拡張可能バリアント型を一般的な言語機能として提供することもできるが、ML の設計者たちは exn を単純に特別扱いすることを選んだ。

[†2] exception 構文は束縛子なので、必要に応じて α 変換をすることで、T_{exn} で既に使われているタグと l が異なることを常に保証できる。

ブクラスのインスタンスは、throw 文（本章の raise と同じ）や try...catch 文（本章の try...with と同じ）の中で使える。新しい例外は、単に Throwable の新しいサブクラスを定義することで宣言できる。

この例外処理機構と ML のそれとの間には実は密接な対応がある。大雑把に言って、Java の例外オブジェクトは、実行時には、クラスを示すタグ（ML の拡張可能バリアントタグに直接対応する）と、インスタンス変数のレコード（そのタグによってラベル付けされた追加情報に対応する）とで表現される。

幾つかの点で Java の例外は ML よりさらに少し進んでいる。一つには、**サブクラス**の順序付けによって作られる、例外タグの上の自然な半順序がある。例外 T のためのハンドラは、実際にはクラス T または T の任意のサブクラスのオブジェクトを運ぶ例外を捕捉する。他にも Java は、アプリケーションプログラムが捕捉して回復を試みようとするかもしれない**例外**（Throwable のサブクラスである組み込みクラス Exception のサブクラス）と、通常は単に実行を終了すべき深刻な状況を示す**エラー**（同じく Throwable のサブクラスである Error のサブクラス）を区別している。これらの鍵となる違いは型検査規則にある。メソッドは送出する可能性のある例外を明示的に宣言しなければならないが、エラーは宣言しなくてもよい。

演習 14.3.1. [★★★]　上記の (4) の方針である拡張可能バリアント型の説明はかなり非形式的である。正確にする方法を示せ。 □

演習 14.3.2. [★★★★]　Java の例外（Exception のサブクラスであるもの）は ML の例外（あるいは本章で定義した例外）より少し厳密に制御されていると述べた。すなわち、メソッドが送出する可能性のあるすべての例外は、メソッドの型において宣言されていなければならない。演習 14.3.1 の解答を拡張して、関数の型が引数と返り値の型だけでなく送出しうる例外の集合も示すようにせよ。その体系が型安全であることを示せ。 □

演習 14.3.3. [★★★]　他の制御構文の多くは、この章で見てきた方法と同様の方法で形式化できる。Scheme の "call with current continuation"（call/cc）演算子（Clinger, Friedman, and Wand [103]; Kelsey, Clinger, and Rees [252]; Dybvig [142]; Friedman, Wand, and Haynes [163]）に詳しい読者は、**T-継続**（型 T の引数を受け取る継続）の型 Cont T に基づいた型付け規則の形式化に挑戦すると面白いだろう。 □

第III部

部分型付け

第 15 章

部分型付け

本章で扱う体系は $\lambda_{<:}$、すなわち部分型付け（図 15-1）とレコード（図 15-3）を含む単純型付きラムダ計算である。対応する OCaml 実装は `rcdsub` である。（一部の例では数も扱う。それらの検査には `fullsub` が必要である。）

ここ数章では、単純型付きラムダ計算の枠組みの中でさまざまな言語機能が型付けに関してどのように振る舞うかを学んできた。この章ではより根本的な拡張である**部分型付け**（ときとして**部分型多相**と呼ばれる）を扱う。これまで学習してきた機能は互いにおおよそ独立に定式化することができた。しかし部分型付けは横断的な拡張であり、他の多くの言語機能と自明でない形で互いに影響し合う。

部分型付けは、**オブジェクト指向**言語に特徴的に見られるもので、しばしばオブジェクト指向の本質的な機能であると考えられている。この関連性は詳しくは第 18 章で扱うこととし、ひとまず関数とレコードだけという簡潔な状況に限定して部分型付けを議論する。この二つに限定しても、興味深い問題のほとんどが出現する。15.5 節では部分型付けとこれまでの章で見てきた他の機能を組み合わせることについて議論する。最後の 15.6 節では、部分型付けのより詳細な意味論について考える。その意味論では、部分型付けは実行時の**型強制**を挿入することに対応する。

15.1 包摂

部分型付けがなければ、単純型付きラムダ計算の規則は融通が利かないものとなりうる。引数の型が関数の定義域の型と完全に一致していなければならないということを型システムが強く要求するせいで、プログラマが見ればよく振る舞うことが明らかなプログラムが型検査器に拒否されてしまうようになる。例えば、関数適用の型付け規則を思い出そう。

$$\frac{\Gamma \vdash t_1 : T_{11} \to T_{12} \qquad \Gamma \vdash t_2 : T_{11}}{\Gamma \vdash t_1\ t_2 : T_{12}} \qquad \text{(T-APP)}$$

この規則によると、よく振る舞う項

```
(λr:{x:Nat}. r.x) {x=0,y=1}
```

は型付け不可能である。なぜなら、引数の型が `{x:Nat,y:Nat}` である一方、関数は `{x:Nat}` を受け付けるからである。しかし明らかに、この関数は引数が `x` のフィールドを持つレコードであることを要求しているにすぎず、他のフィールドを持つかどうかは気にしていない。さらに、このことは関数の型だけからわかる。つまり、`x` 以外のフィールドを用いないことを確かめるために、その本体を見る必要はない。`{x:Nat}` の型を期待している関数に型 `{x:Nat,y:Nat}` の引数を渡すことは**常に**安全である。

部分型付けの目標は、上記のような項を許すように型付け規則を改良することである。これを達成するために、ある型が他の型より情報を持っているという直観を定式化する。すなわち、型 S の任意の項が、型 T の項が期待されている文脈で安全に使用可能であるということを、S は T の**部分型**であるといい、これを S <: T と書く。このような部分型付けの捉え方はしばしば**安全代入の原則**と呼ばれる。

より単純な直観は、S <: T を「S で記述されるすべての値は T でも記述される」と読むことである。すなわち、S のすべての要素からなる集合は、T のすべての要素からなる集合の部分集合になっている、ということである。15.6 節で、より洗練された別の解釈で部分型付けを扱うことが有用な場合があるのを見る。しかし、大概の目的には、この**部分集合意味論**で十分である。

型付け関係と部分型関係をつなぐために、新しい型付け規則を加える。いわゆる**包摂**（subsumption）規則である。

$$\frac{\Gamma \vdash \mathtt{t} : \mathtt{S} \qquad \mathtt{S} <: \mathtt{T}}{\Gamma \vdash \mathtt{t} : \mathtt{T}} \qquad \text{(T-SUB)}$$

この規則は、S <: T ならば S 型の要素 t はすべて T の要素でもあると言っている。例えば、{x:Nat,y:Nat} <: {x:Nat} となるように部分型関係を定義したとすると、規則 T-SUB を用いて ⊢ {x=0,y=1} : {x:Nat} を導ける。先ほどの動機付けを与えた例が型検査を通るようにするためには、これが必要となる。

15.2 部分型関係

部分型関係は、S <: T という形式の部分型付け判断式を導出するための推論規則の集まりとして定式化される。S <: T は「S は T の部分型である」もしくは「T は S の上位型である」と読む。ここで、それぞれの型の形（関数型、レコード型など）を個別に考えることにする。すなわちそれぞれの形に関して、その形の型の要素が期待されているところで、他の型の要素の使用を許しても安全であるという状況を定式化するような規則を一つ以上導入する。

個別の型構築子の規則に取り掛かる前に、二つの一般的な規定を作る。一つ目は、部分型付けが反射的でなければならないということである。

$$\mathtt{S} <: \mathtt{S} \qquad \text{(S-REFL)}$$

二つ目は、部分型付けは推移的でなければならないということである。

$$\frac{\mathtt{S} <: \mathtt{U} \qquad \mathtt{U} <: \mathtt{T}}{\mathtt{S} <: \mathtt{T}} \qquad \text{(S-TRANS)}$$

これらの規則は、安全代入の直観から直接得られる。

さて既に述べたように、レコード型について、型 $\mathtt{T} = \{\mathtt{l}_1\mathtt{:T}_1 \ldots \mathtt{l}_n\mathtt{:T}_n\}$ が持つフィールドが型 $\mathtt{S} = \{\mathtt{k}_1\mathtt{:S}_1 \ldots \mathtt{k}_m\mathtt{:S}_m\}$ のものより少なければ S を T の部分型とみなしたい。具体的には、レコード型の終端のフィールドを幾つか「忘れ」ても安全だということである。この直観はいわゆる**幅部分型付け**規則で表せる。

$$\{\mathtt{l}_i\mathtt{:T}_i\ ^{i\in 1..n+k}\} <: \{\mathtt{l}_i\mathtt{:T}_i\ ^{i\in 1..n}\} \qquad \text{(S-RCDWIDTH)}$$

フィールドの数が**多い**ほうが「より小さい」型（部分型）であるのは、意外に思えるかもしれない。これを最も簡単に理解する方法は、レコード型に対して 11.8 節のときよりも自由な見方を採用することである。すなわち、レコード型 {x:Nat} が「**少なくとも**型 Nat のフィールド x を持つすべてのレコードの集合」を表していると捉えるのである。{x=3} や {x=5} のような値はすべてこの型の要素であり、{x=3, y=100} や {x=3,a=true,b=true} のような値も同様である。同様に、レコード型 {x:Nat, y:Nat} は**少なくとも**フィールド x と y を持ち、その両方が型 Nat であるすべてのレコードを表している。{x=3,y=100} や {x=3,y=100,z=true} はこの型の値である。しかし、{x=3} はそうではないし、{x=3,a=true,b=true} もそうではない。したがって、二つ目の型に属

する値の集合は、一つ目の型に属する値の集合の真の部分集合になっている。フィールドが多いレコードは、より要求の厳しい（つまり、より情報量のある）仕様を構成する。そのため、値のより小さな集合を表現していることになる。

幅部分型付け規則が適用できるのは、共通のフィールドの型が全く同じであるようなレコード型だけである。しかし、二つのレコード中で対応するフィールドの型がすべて部分型関係にある限りは、個々のフィールドの型が異なることを許しても安全である。**深さ部分型付け**規則がこの直観を表している。

$$\frac{\text{各 } i \text{ に対して } \mathtt{S}_i <: \mathtt{T}_i}{\{\mathtt{l}_i\mathtt{:S}_i\ ^{i\in 1..n}\} <: \{\mathtt{l}_i\mathtt{:T}_i\ ^{i\in 1..n}\}} \qquad \text{(S-RcdDepth)}$$

次の部分型付け導出は、S-RcdWidth と S-RcdDepth を併せて用いて、入れ子のレコード型 {x:{a:Nat,b:Nat},y:{m:Nat}} が {x:{a:Nat},y:{}} の部分型であることを示している。

$$\frac{\dfrac{}{\mathtt{\{a:Nat,b:Nat\}} <: \mathtt{\{a:Nat\}}}\ \text{S-RcdWidth} \qquad \dfrac{}{\mathtt{\{m:Nat\}} <: \mathtt{\{\}}}\ \text{S-RcdWidth}}{\mathtt{\{x:\{a:Nat,b:Nat\},y:\{m:Nat\}\}} <: \mathtt{\{x:\{a:Nat\},y:\{\}\}}}\ \text{S-RcdDepth}$$

S-RcdDepth を用いて（上の例のようにレコードのすべてのフィールドを変更するのではなく）一つのフィールドだけの型を変更したい場合は、残りのフィールドに対して S-Refl を用いて自明な部分型付け導出を得ればよい。

$$\frac{\dfrac{}{\mathtt{\{a:Nat,b:Nat\}} <: \mathtt{\{a:Nat\}}}\ \text{S-RcdWidth} \qquad \dfrac{}{\mathtt{\{m:Nat\}} <: \mathtt{\{m:Nat\}}}\ \text{S-Refl}}{\mathtt{\{x:\{a:Nat,b:Nat\},y:\{m:Nat\}\}} <: \mathtt{\{x:\{a:Nat\},y:\{m:Nat\}\}}}\ \text{S-RcdDepth}$$

推移律の規則 S-Trans を用いて、幅部分型付けと深さ部分型付けを組み合わせることもできる。例えば、あるフィールドの型を昇格させつつ別のフィールドを削除することで、上位型を得られる。

$$\frac{\dfrac{}{\begin{array}{c}\mathtt{\{x:\{a:Nat,b:Nat\},y:\{m:Nat\}\}}\\ <: \mathtt{\{x:\{a:Nat,b:Nat\}\}}\end{array}}\ \text{S-RcdWidth} \qquad \dfrac{\dfrac{}{\begin{array}{c}\mathtt{\{a:Nat,b:Nat\}}\\ <: \mathtt{\{a:Nat\}}\end{array}}\ \text{S-RcdWidth}}{\begin{array}{c}\mathtt{\{x:\{a:Nat,b:Nat\}\}}\\ <: \mathtt{\{x:\{a:Nat\}\}}\end{array}}\ \text{S-RcdDepth}}{\mathtt{\{x:\{a:Nat,b:Nat\},y:\{m:Nat\}\}} <: \mathtt{\{x:\{a:Nat\}\}}}\ \text{S-Trans}$$

レコードの部分型付けの最後の規則は、レコードをどのように利用すれば安全かということに関して、フィールドの順序は影響しないという観察から得られる。というのも、いったん構築してしまったレコードに対して唯一**できること**（つまり、フィールドを射影すること）は、フィールドの順序とは無関係だからである。

$$\frac{\{\mathtt{k}_j\mathtt{:S}_j\ ^{j\in 1..n}\} \text{ は } \{\mathtt{l}_i\mathtt{:T}_i\ ^{i\in 1..n}\} \text{ の並べ替え(permutation)である}}{\{\mathtt{k}_j\mathtt{:S}_j\ ^{j\in 1..n}\} <: \{\mathtt{l}_i\mathtt{:T}_i\ ^{i\in 1..n}\}} \qquad \text{(S-RcdPerm)}$$

例えば、S-RcdPerm から {c:Bool,b:Bool,a:Nat} が {a:Nat,b:Bool,c:Bool} の部分型であること（またその逆も）がわかる。（これは部分型関係が反対称的で**ない**ことを意味する。）

S-RcdPerm を S-RcdWidth と S-Trans と組み合わせて用いると、レコード型の後方のフィールドだけでなく、任意の位置のフィールドを削除できる。

演習 15.2.1. [★] {x:Nat,y:Nat,z:Nat} が {y:Nat} の部分型であることの導出を書け。 □

S-RcdWidth、S-RcdDepth、S-RcdPerm は、レコードの使用に関してそれぞれ異なる種類の柔軟性を実現する。議論が目的ならば、これらを三つの別の規則として説明するのが有用である。具体的には、これらの規則を一部だけしか許さない言語がある。例えば Abadi と Cardelli の**オブジェクト計算** [4] の変種のほとんどは、深さ部分型付けを省略している。しかし、実装が目的ならば、これらの規則を組み合わせて、三つ全部を同時に行う単一の大規則にしておくほうが便利である。この規則は次の章（164 ページ）で議論する。

我々が扱っているのは、数やレコードだけでなく関数も他の関数に引数として渡せる高階の言語である。したがって、関数型に対する部分型付け規則も与えなければならない。すなわち、どのような状況下であれば、ある型の関数を別の関数型が期待されている文脈で用いても安全であるかを指定しなければならない。

$$\frac{T_1 <: S_1 \qquad S_2 <: T_2}{S_1 \to S_2 <: T_1 \to T_2} \qquad \text{(S-Arrow}^{\dagger 1}\text{)}$$

左側の前提で、引数の型に対して部分型関係の向きが反転している（**反変**という）一方、返り値の型は関数型自身と同じ向きになっている（**共変**という）ことに注意されたい。ここでの直観は次の通りである。型 $S_1 \to S_2$ の関数 f があれば、f は型 S_1 の要素を受け取るとわかる。すなわち、明らかに f は S_1 の任意の部分型 T_1 の要素も受け入れる。さらに f の型から、f が型 S_2 の要素を返すこともわかる。その返り値は、S_2 の任意の上位型 T_2 に属するともみなせる。つまり、型 $S_1 \to S_2$ の任意の関数 f は、型 $T_1 \to T_2$ を持つとみなすこともできる。

別の見方をすると、型 $S_1 \to S_2$ の関数を別の型 $T_1 \to T_2$ が期待されている文脈で用いることを許すのが安全なのは、関数に渡されるどの引数もその関数にとって想定外なものでなく（つまり $T_1 <: S_1$）、かつ、どの返り値も文脈から見て想定外なものではない（つまり $S_2 <: T_2$）場合に限る。

最後に、すべての型の上位型となるような型があると便利である。新しい型定数 Top を導入し、さらに、Top を部分型関係の最大の要素とする規則を導入する。

$$S <: \mathsf{Top} \qquad \text{(S-Top)}$$

Top 型については 15.4 節でさらに議論する。

形式的には、部分型関係はこれまで定義してきた規則に関して閉じた最小の関係である。参照しやすいよう、図 15-1、図 15-2、図 15-3 に、レコードと部分型付けを備える単純型付きラムダ計算のすべての定義を要約する。この章で追加した文法的な形式や規則は網掛けにしてある。反射律と推移律の規則があるので、部分型関係が明らかに**前順序**であることに注意されたい。しかしレコードの並べ替え規則の影響で、この順序は半順序にはならない。異なる型の組で、互いに部分型関係であるようなものが多数存在する。

部分型関係の議論の仕上げとして、本章の最初に挙げた例の型検査が成功するようになったことを確認しよう。ページの端からはみ出ないよう、以下の略記を用いる。

```
f   ≝  λr:{x:Nat}. r.x        Rx   ≝  {x:Nat}
xy  ≝  {x=0,y=1}              Rxy  ≝  {x:Nat,y:Nat}
```

また、数の定数のための普通の型付け規則を仮定する。すると、型付け判断式 ⊢ f xy : Nat は次のように導出を構築できる。

†1 （訳注）「Arrow」は矢印型（arrow type）であることから。

$\to$ $<:$ Top　　　　　　　　　　　　　　　　　λ→（図 9-1）に基づく

構文

t ::=		項：
	x	変数
	λx:T.t	ラムダ抽象
	t t	関数適用
v ::=		値：
	λx:T.t	ラムダ抽象値
T ::=		型：
	Top	最大の型
	T→T	関数の型
Γ ::=		文脈：
	∅	空の文脈
	Γ, x:T	項変数の束縛

評価　　$\boxed{t \longrightarrow t'}$

$$\frac{t_1 \longrightarrow t_1'}{t_1\ t_2 \longrightarrow t_1'\ t_2} \quad \text{(E-APP1)}$$

$$\frac{t_2 \longrightarrow t_2'}{v_1\ t_2 \longrightarrow v_1\ t_2'} \quad \text{(E-APP2)}$$

$$(\lambda x{:}T_{11}.t_{12})\ v_2 \longrightarrow [x \mapsto v_2]t_{12} \quad \text{(E-APPABS)}$$

部分型付け　　$\boxed{S <: T}$

$$S <: S \quad \text{(S-REFL)}$$

$$\frac{S <: U \qquad U <: T}{S <: T} \quad \text{(S-TRANS)}$$

$$S <: Top \quad \text{(S-TOP)}$$

$$\frac{T_1 <: S_1 \qquad S_2 <: T_2}{S_1 \to S_2 <: T_1 \to T_2} \quad \text{(S-ARROW)}$$

型付け　　$\boxed{\Gamma \vdash t : T}$

$$\frac{x{:}T \in \Gamma}{\Gamma \vdash x : T} \quad \text{(T-VAR)}$$

$$\frac{\Gamma, x{:}T_1 \vdash t_2 : T_2}{\Gamma \vdash \lambda x{:}T_1.t_2 : T_1 \to T_2} \quad \text{(T-ABS)}$$

$$\frac{\Gamma \vdash t_1 : T_{11} \to T_{12} \qquad \Gamma \vdash t_2 : T_{11}}{\Gamma \vdash t_1\ t_2 : T_{12}} \quad \text{(T-APP)}$$

$$\frac{\Gamma \vdash t : S \qquad S <: T}{\Gamma \vdash t : T} \quad \text{(T-SUB)}$$

▲図 15-1.　部分型付けを持つ単純型付きラムダ計算（$\lambda_{<:}$）

$$\dfrac{\dfrac{\vdots}{\vdash f : Rx \to Nat} \qquad \dfrac{\dfrac{\dfrac{}{\vdash 0 : Nat} \quad \dfrac{}{\vdash 1 : Nat}}{\vdash xy : Rxy}\,\text{T-RCD} \qquad \dfrac{}{Rxy <: Rx}\,\text{S-RCDWIDTH}}{\vdash xy : Rx}\,\text{T-SUB}}{\vdash f\,xy : Nat}\,\text{T-APP}$$

演習 15.2.2. [★]　これが型付け判断式 $\vdash$ f xy : Nat の唯一の導出か。 □

演習 15.2.3. [★]　(1) {a:Top,b:Top} は異なる上位型を幾つ持つか。(2) 部分型関係の無限降下列を見つけられるか。すなわち、無限列 S_0、S_1、…で、S_{i+1} が S_i の部分型であるようなものはあるか。(3) 無限上昇列はどうだろうか。 □

演習 15.2.4. [★]　他のすべての型の部分型となるような型はあるだろうか。矢印型で、他のすべての矢印型の上位型となるような型はあるだろうか。 □

$\rightarrow$ {}　　λ→（図 9-1）の拡張

新しい構文形式

$$\begin{array}{llll} \mathtt{t} & ::= & \ldots & \text{項：} \\ & & \{\mathtt{l}_i\mathtt{=t}_i\ ^{i\in 1..n}\} & \text{レコード} \\ & & \mathtt{t.l} & \text{射影} \\ \mathtt{v} & ::= & \ldots & \text{値：} \\ & & \{\mathtt{l}_i\mathtt{=v}_i\ ^{i\in 1..n}\} & \text{レコードの値} \\ \mathtt{T} & ::= & \ldots & \text{型：} \\ & & \{\mathtt{l}_i\mathtt{:T}_i\ ^{i\in 1..n}\} & \text{レコードの型} \end{array}$$

新しい評価規則　　$\boxed{\mathtt{t} \longrightarrow \mathtt{t}'}$

$$\{\mathtt{l}_i\mathtt{=v}_i\ ^{i\in 1..n}\}\mathtt{.l}_j \longrightarrow \mathtt{v}_j \qquad \text{(E-ProjRcd)}$$

$$\frac{\mathtt{t}_1 \longrightarrow \mathtt{t}_1'}{\mathtt{t}_1\mathtt{.l} \longrightarrow \mathtt{t}_1'\mathtt{.l}} \qquad \text{(E-Proj)}$$

$$\frac{\mathtt{t}_j \longrightarrow \mathtt{t}_j'}{\begin{array}{l}\{\mathtt{l}_i\mathtt{=v}_i\ ^{i\in 1..j-1}\mathtt{,l}_j\mathtt{=t}_j\mathtt{,l}_k\mathtt{=t}_k\ ^{k\in j+1..n}\} \\ \longrightarrow \{\mathtt{l}_i\mathtt{=v}_i\ ^{i\in 1..j-1}\mathtt{,l}_j\mathtt{=t}_j'\mathtt{,l}_k\mathtt{=t}_k\ ^{k\in j+1..n}\}\end{array}} \qquad \text{(E-Rcd)}$$

新しい型付け規則　　$\boxed{\Gamma \vdash \mathtt{t} : \mathtt{T}}$

$$\frac{\text{各 } i \text{ に対して } \Gamma \vdash \mathtt{t}_i : \mathtt{T}_i}{\Gamma \vdash \{\mathtt{l}_i\mathtt{=t}_i\ ^{i\in 1..n}\} : \{\mathtt{l}_i\mathtt{:T}_i\ ^{i\in 1..n}\}} \qquad \text{(T-Rcd)}$$

$$\frac{\Gamma \vdash \mathtt{t}_1 : \{\mathtt{l}_i\mathtt{:T}_i\ ^{i\in 1..n}\}}{\Gamma \vdash \mathtt{t}_1\mathtt{.l}_j : \mathtt{T}_j} \qquad \text{(T-Proj)}$$

▲ 図 15-2.　レコード（図 11-7 と同じ）

$\rightarrow$ {} <:　　λ<:（図 15-1）と単純なレコードの規則（図 15-2）の拡張

新しい部分型付け規則　　$\boxed{\mathtt{S} <: \mathtt{T}}$

$$\{\mathtt{l}_i\mathtt{:T}_i\ ^{i\in 1..n+k}\} <: \{\mathtt{l}_i\mathtt{:T}_i\ ^{i\in 1..n}\} \qquad \text{(S-RcdWidth)}$$

$$\frac{\text{各 } i \text{ に対して } \mathtt{S}_i <: \mathtt{T}_i}{\{\mathtt{l}_i\mathtt{:S}_i\ ^{i\in 1..n}\} <: \{\mathtt{l}_i\mathtt{:T}_i\ ^{i\in 1..n}\}} \qquad \text{(S-RcdDepth)}$$

$$\frac{\{\mathtt{k}_j\mathtt{:S}_j\ ^{j\in 1..n}\} \text{ は } \{\mathtt{l}_i\mathtt{:T}_i\ ^{i\in 1..n}\} \text{ の並べ替えである}}{\{\mathtt{k}_j\mathtt{:S}_j\ ^{j\in 1..n}\} <: \{\mathtt{l}_i\mathtt{:T}_i\ ^{i\in 1..n}\}} \qquad \text{(S-RcdPerm)}$$

▲ 図 15-3.　レコードと部分型付け

演習 15.2.5. [★★]　11.6 節で説明した直積型の構築子 $\mathtt{T}_1 \times \mathtt{T}_2$ で計算体系を拡張するとする。以下の部分型付け規則を追加することは自然である。

$$\frac{\mathtt{S}_1 <: \mathtt{T}_1 \qquad \mathtt{S}_2 <: \mathtt{T}_2}{\mathtt{S}_1 \times \mathtt{S}_2 <: \mathtt{T}_1 \times \mathtt{T}_2} \qquad \text{(S-ProdDepth)}$$

これはレコードにおける S-RcdDepth に対応する。直積型の幅部分型付け規則も追加することは、良いアイディアだろうか。

$$\mathtt{T}_1 \times \mathtt{T}_2 <: \mathtt{T}_1 \qquad \text{(S-ProdWidth)}$$

□

15.3 部分型付けと型付けに関する性質

部分型付けを持つラムダ計算の定義を決定したが、これが意味のある定義であることを検証するために（特に、単純型付きラムダ計算の保存と進行の定理が部分型付けの存在下でもやはり成り立つことを検証するために）、やらなければならないことがある。

演習 15.3.1. ［推奨, ★★］ 読み進む前に、どこで問題が起きるか予想してみよう。具体的には、部分型関係の定義でミスを犯し、上の定義に加えて誤った部分型付け規則を入れてしまったとする。この体系のどの性質が壊れてしまう可能性があるだろうか。一方、部分型付け規則の一つを**省略した**とすると、いずれかの性質を壊してしまうだろうか。 □

部分型関係のある重要な性質を述べることから始めよう。単純型付きラムダ計算の型付け関係の逆転補題（補題 9.3.1）に似たものである。もし、ある型 S が矢印型の部分型であると知っていたとすると、部分型付けの逆転補題から S そのものが矢印型でなければならないとわかる。さらに矢印の左側は（反変的に）関連しており、また右側は（共変的に）関連していなければならないこともわかる。同様の考えを、S がレコード型の部分型であるとわかっている場合に適用する。S がより多くのフィールド（S-RcdWidth）をある順序（S-RcdPerm）で持っており、共通のフィールドの型は部分型関係（S-RcdDepth）にあることがわかる。

補題 15.3.2. ［**部分型関係の逆転**］

(1) もし $S <: T_1 \to T_2$ であれば、S は $S_1 \to S_2$ という形式であり、$T_1 <: S_1$ かつ $S_2 <: T_2$ である。
(2) もし $S <: \{l_i : T_i\ ^{i \in 1..n}\}$ であれば、S は $\{k_j : S_j\ ^{j \in 1..m}\}$ という形式である。また、S は少なくともラベル $\{l_i\ ^{i \in 1..n}\}$ を持つ（すなわち $\{l_i\ ^{i \in 1..n}\} \subseteq \{k_j\ ^{j \in 1..m}\}$ となる）。さらに、共通のラベル $l_i = k_j$ では $S_j <: T_i$ となる。 □

［証明］ 演習［推奨, ★★］。 □

型が評価の際に保存されていることを証明するために、型付け関係に関する逆転補題から始めよう（単純型付きラムダ計算の補題 9.3.1 を参照）。補題の最も一般的な形式を述べることはせず、ここでは下記の保存定理の証明で実際に必要になるケースだけを与えることにする。（一般的な形式は次の章の定義 16.2.2 においてアルゴリズム的部分型関係から読み取れる。）

補題 15.3.3.

(1) もし $\Gamma \vdash \lambda x{:}S_1.\ s_2 : T_1 \to T_2$ であれば、$T_1 <: S_1$ かつ $\Gamma, x{:}S_1 \vdash s_2 : T_2$ である。
(2) もし $\Gamma \vdash \{k_a{=}s_a\ ^{a \in 1..m}\} : \{l_i{:}T_i\ ^{i \in 1..n}\}$ であれば、$\{l_i\ ^{i \in 1..n}\} \subseteq \{k_a\ ^{a \in 1..m}\}$ かつ、共通のラベル $k_a = l_i$ それぞれに対して $\Gamma \vdash s_a : T_i$ が成り立つ。 □

［証明］ 型付け導出に関する単純な帰納法。T-Sub の場合には補題 15.3.2 を用いる。 □

次に、型付け関係に対する代入補題が必要である。この補題の言明は、単純型付きラムダ計算のもの（補題 9.3.8）から変更されておらず、その証明もほとんど同一である。

補題 15.3.4. ［**代入**］ もし $\Gamma, x{:}S \vdash t : T$ かつ $\Gamma \vdash s : S$ であれば、$\Gamma \vdash [x \mapsto s]t : T$ が成り立つ。 □

［証明］ 型付けの導出の深さに関する帰納法による。T-Sub とレコードの型付け規則 T-Rcd と T-Proj の場合を新しく扱う必要があるが、これは帰納法の仮定を単純に適用すればよい。残りは補題 9.3.8 の証明と同様である。 □

ここで、保存定理は以前と同じ言明を持つ。しかしその証明は、部分型付けの影響で数か所がやや複雑になっている。

定理 15.3.5.［保存］ もし $\Gamma \vdash \mathtt{t} : \mathtt{T}$ かつ $\mathtt{t} \longrightarrow \mathtt{t}'$ であれば、$\Gamma \vdash \mathtt{t}' : \mathtt{T}$ □

［証明］ 型付け導出に関する単純な帰納法。ほとんどの場合は単純型付きラムダ計算の型保存の証明（定理 9.3.9 の証明）と同様である。レコードの型付け規則の場合と、包摂の場合を新しく扱う必要がある。

T-VAR の場合 $\mathtt{t} = \mathtt{x}$
起こりえない（変数の評価規則はない）。

T-ABS の場合 $\mathtt{t} = \lambda\mathtt{x}{:}\mathtt{T}_1\,.\,\mathtt{t}_2$
起こりえない（t は既に値である）。

T-APP の場合 $\mathtt{t} = \mathtt{t}_1\ \mathtt{t}_2$ $\quad \Gamma \vdash \mathtt{t}_1 : \mathtt{T}_{11}{\to}\mathtt{T}_{12}$ $\quad \Gamma \vdash \mathtt{t}_2 : \mathtt{T}_{11}$ $\quad \mathtt{T} = \mathtt{T}_{12}$
図 15-1 と図 15-2 の評価規則から、$\mathtt{t} \longrightarrow \mathtt{t}'$ を導ける三つの規則を読み取れる。E-APP1、E-APP2、E-APPABS である。場合分けを行う。

E-APP1 の場合 $\mathtt{t}_1 \longrightarrow \mathtt{t}_1'$ $\quad \mathtt{t}' = \mathtt{t}_1'\ \mathtt{t}_2$
帰納法の仮定と T-APP から結論が導かれる。

E-APP2 の場合 $\mathtt{t}_1 = \mathtt{v}_1$ $\quad \mathtt{t}_2 \longrightarrow \mathtt{t}_2'$ $\quad \mathtt{t}' = \mathtt{v}_1\ \mathtt{t}_2'$
同様。

E-APPABS の場合 $\mathtt{t}_1 = \lambda\mathtt{x}{:}\mathtt{S}_{11}\,.\,\mathtt{t}_{12}$ $\quad \mathtt{t}_2 = \mathtt{v}_2$ $\quad \mathtt{t}' = [\mathtt{x} \mapsto \mathtt{v}_2]\mathtt{t}_{12}$
補題 15.3.3 (1) から、$\mathtt{T}_{11}$ <: $\mathtt{S}_{11}$ と $\Gamma, \mathtt{x}{:}\mathtt{S}_{11} \vdash \mathtt{t}_{12} : \mathtt{T}_{12}$ である。T-SUB より、$\Gamma \vdash \mathtt{t}_2 : \mathtt{S}_{11}$ である。これと代入補題（補題 15.3.4）より、$\Gamma \vdash \mathtt{t}' : \mathtt{T}_{12}$ を得る。

T-RCD の場合 $\mathtt{t} = \{\mathtt{l}_i{=}\mathtt{t}_i\ ^{i\in 1..n}\}$ $\quad$ 各 i に対して $\Gamma \vdash \mathtt{t}_i : \mathtt{T}_i$
$\mathtt{T} = \{\mathtt{l}_i{:}\mathtt{T}_i\ ^{i\in 1..n}\}$
左辺がレコードであるような評価規則は E-RCD のみである。この規則の前提より、あるフィールド $\mathtt{t}_j$ について $\mathtt{t}_j \longrightarrow \mathtt{t}_j'$ を得る。帰納法の仮定（対応する仮定 $\Gamma \vdash \mathtt{t}_j : \mathtt{T}_j$ に適用される）と T-RCD より結論が導かれる。

T-PROJ の場合 $\mathtt{t} = \mathtt{t}_1.\mathtt{l}_j$ $\quad \Gamma \vdash \mathtt{t}_1 : \{\mathtt{l}_i{:}\mathtt{T}_i\ ^{i\in 1..n}\}$ $\quad \mathtt{T} = \mathtt{T}_j$
図 15-1 と図 15-2 の評価規則から $\mathtt{t} \longrightarrow \mathtt{t}'$ を導く規則は二つある。E-PROJ と E-PROJRCD である。

E-PROJ の場合 $\mathtt{t}_1 \longrightarrow \mathtt{t}_1'$ $\quad \mathtt{t}' = \mathtt{t}_1'.\mathtt{l}_j$
帰納法の仮定と T-PROJ から結論が導かれる。

E-PROJRCD の場合 $\mathtt{t}_1 = \{\mathtt{k}_a{=}\mathtt{v}_a\ ^{a\in 1..m}\}$ $\quad \mathtt{l}_j = \mathtt{k}_b$ $\quad \mathtt{t}' = \mathtt{v}_b$
補題 15.3.3 (2) から、$\{\mathtt{l}_i\ ^{i\in 1..n}\} \subseteq \{\mathtt{k}_a\ ^{a\in 1..m}\}$ と、各 $\mathtt{k}_a = \mathtt{l}_i$ に対して $\Gamma \vdash \mathtt{v}_a : \mathtt{T}_i$ を得る。特に、所望の通り、$\Gamma \vdash \mathtt{v}_b : \mathtt{T}_j$ となる。

T-SUB の場合 $\mathtt{t} : \mathtt{S}$ $\quad \mathtt{S}$ <: $\mathtt{T}$
帰納法の仮定より $\Gamma \vdash \mathtt{t}' : \mathtt{S}$。T-SUB より $\Gamma \vdash \mathtt{t}' : \mathtt{T}$。 □

正しく型付けされた項が行き詰まり状態にならないことを証明するために、（第 9 章のように）標準形補題から始めよう。それによって矢印型やレコード型に属する値が取りうる形がわかる。

補題 15.3.6.［標準形］

(1) もし v が型 $\mathtt{T}_1{\to}\mathtt{T}_2$ の閉じた値であれば、v は $\lambda\mathtt{x}{:}\mathtt{S}_1\,.\,\mathtt{t}_2$ という形である。

(2) もし v が型 $\{l_i\mathtt{:}T_i\ ^{i\in 1..n}\}$ の閉じた値であれば、v は $\{k_a\mathtt{=}v_a\ ^{a\in 1..m}\}$ という形を持ち、$\{l_i\ ^{i\in 1..n}\} \subseteq \{k_a\ ^{a\in 1..m}\}$ となる。 □

[証明] 演習[推奨, ★★★]。 □

進行定理とその証明は、今や単純型付きラムダ計算の際に扱ったものと非常に近い。部分型付けを扱う難しい部分のほとんどは標準形補題に押し込められており、ここではほんのわずかな変更のみが必要である。

定理 15.3.7. [進行] t が閉じた、正しく型付けされた項であれば、t は値であるか、もしくは、ある t′ について t ⟶ t′ □

[証明] 型付け導出に関する単純な帰納法による。変数の場合は起こりえない（t が閉じているため）。ラムダ抽象の場合、抽象は値であるため、直ちに従う。残りの場合はより興味深い。

T-APP の場合 $t = t_1\ t_2$ $\vdash t_1 : T_{11}\to T_{12}$ $\vdash t_2 : T_{11}$ $T = T_{12}$

帰納法の仮定により、t_1 は値であるか、もしくは評価を 1 ステップ進められる。t_2 も同様である。もし t_1 の評価を進められるなら、規則 E-APP1 を t へ適用する。もし t_1 が値で t_2 の評価を 1 ステップ進められる場合、規則 E-APP2 を適用する。最後に、どちらも値である場合、標準形補題（補題 15.3.6）により t_1 は $\lambda x\mathtt{:}S_{11}.t_{12}$ という形を持つので、規則 E-APPABS を t に適用する。

T-RCD の場合 $t = \{l_i\mathtt{=}t_i\ ^{i\in 1..n}\}$ 各 $i \in 1..n$ について $\vdash t_i : T_i$
$T = \{l_i\mathtt{:}T_i\ ^{i\in 1..n}\}$

帰納法の仮定により、各 t_i は値であるか評価を 1 ステップ進められる。もしすべてが値であれば、t は値である。一方、一つでも評価を進められれば、規則 E-RCD を t に適用する。

T-PROJ の場合 $t = t_1.l_j$ $\vdash t_1 : \{l_i\mathtt{:}T_i\ ^{i\in 1..n}\}$ $T = T_j$

帰納法の仮定により、t_1 は値であるか評価を 1 ステップ進められる。もし t_1 が評価を進められるなら、t も評価を進められる（E-PROJ による）。もし t_1 が値であれば、標準形補題（補題 15.3.6）により、t_1 は $\{k_a\mathtt{=}v_a\ ^{a\in 1..m}\}$ という形式であり、$\{l_i\ ^{i\in 1..n}\} \subseteq \{k_a\ ^{a\in 1..m}\}$ かつ、すべての $l_i = k_a$ に対して $\vdash v_a : T_i$ である。特に l_j は t_1 のラベル $\{k_a\ ^{a\in 1..m}\}$ に含まれる。このことから、規則 E-PROJRCD により、t そのものが評価ステップを進められる。

T-SUB の場合 $\Gamma \vdash t : S$ $S <: T$

帰納法の仮定より直ちに従う。 □

15.4 Top 型と Bottom 型

最大の型 Top は、部分型付けを含む単純型付きラムダ計算に必要不可欠なものではない。型システムの性質を壊さずに取り除ける。しかし、ほとんどの形式化には含まれている。これには幾つか理由がある。一つ目は、Top がオブジェクト指向言語のほとんどで見受けられる `Object` 型に対応しているからである。二つ目は、部分型とパラメータ多相を組み合わせた、より洗練された体系にとって、Top を組み込むことが都合がよいからである。例えば System $F_{<:}$（第 26 章、第 28 章）では Top の存在のおかげで、有界量化から普通の非有界量化を取り戻すことができ、型システムが簡素なものになる。実際、$F_{<:}$ はレコード型すら表現でき、説明がもっと簡略になる（少なくとも理論研究の目的では）。この表現は Top なしでは不可能である。最後の理由は、Top の振る舞いは単純で、実例でもしばしば役に立つものなので、省く理由があまりないためである。

→ <: Bot　　　　λ<:（図 15-1）の拡張

新しい構文形式		新しい部分型付け規則	S <: T
T ::= ...	型：	Bot <: T	(S-BOT)
Bot	最小の型		

▲図 15-4.　Bottom 型

では、この部分型関係に最小の要素、すなわちすべての型の部分型である型 Bot も完備することはできるだろうか？ 可能というのが答えだ。図 15-4 はこの拡張の定式化である。

まず注意したいのは、Bot は空ということである。つまり、型 Bot の閉じた値は存在しない。仮に値があるとすると（それを v とする)、包摂規則と S-BOT より ⊢ v : Top→Top を導くことができ、ここから標準形補題（補題 15.3.6、この補題はこの拡張の下でも成り立つ）により、v は、ある S_1 と S_2 に対して $\lambda x{:}S_1.t_2$ という形式を取らなければならないことがわかる。一方、包摂規則により ⊢ v : {} であるから、標準形補題により v はレコードでなければならない。構文から、明らかに v が関数でありながらレコードでもあることはありえない。よって、⊢ v : Bot であるという仮定が矛盾を導いたことがわかる。

Bot が空であるからといって、役に立たないということにはならない。むしろ Bot は、ある種の操作（特に、例外の送出や継続の呼び出し）が戻ってこないはずであるという事実を表現するのに大変都合がよい方法である。そういう式に型 Bot を与えると、良い効果が二つある。一つ目は、結果が期待されていないことをプログラマに伝えられることである（仮に式が結果を返したとしたら、それは型 Bot の値であるから)。二つ目は、そのような式は、どんな型の値が期待されている文脈でも安全に使用できることを型検査器に伝えられることである。例えば、第 14 章で説明した例外を投げる項 error に型 Bot を与えたら、次のような項

```
λx:T.
  if <check that x is reasonable> then
    <compute result>
  else
    error
```

は、正しく型付けされる。なぜなら、正常な場合の返り値の型が何であれ、項 error には常に同じ型を包摂により与えることができ、したがって T-IF が要求する通り if の両方の分岐の型が一致するからである[†2]。

残念ながら、Bot が存在すると型システムの型検査器を作る問題は著しく複雑になる。部分型付けを持つ言語の単純な型付けアルゴリズムでは、「適用 t_1 t_2 が正しく型付けされるなら、t_1 は矢印型を持たないといけない」というような推論に頼る必要がある。Bot の存在下では、この推論を改良して、「t_1 t_2 が正しく型付けされるなら、t_1 は矢印型か型 Bot を持たないといけない」と、より細分化しなければならない（このことは 16.4 節で説明する)。有界量化を持つ体系ではいっそう複雑になる（28.8 節を見よ)。

この複雑さから、Bot を追加することが Top を追加することより重大な一歩であるということがわかる。これ以後、本書では、我々が考える体系からは Bot を取り除いておこう。

[†2] ML のように多相性のある言語では、error やそれに類する構文の返り値の型として ∀X.X を用いることもできる。異なる手段だが、これにより Bot と同じ効果が得られる。任意の型に**昇格**できる型を error に与える代わりに、任意の型に**具体化**できる型スキームを与えるのである。これら二つの解決策は異なる基礎に基づいているが、非常に似たものである。特に、型 ∀X.X も空である。

15.5 部分型付けとその他の機能

部分型付けを持つ単純な計算を本格的なプログラミング言語へ拡張していく際、新しく機能を導入するごとに、それが部分型付けとどのように干渉するかを慎重に検討しなければならない。本節ではこの視点で、これまでに見てきた幾つかの機能を考えていく[†3]。後の章では、部分型付けとある種の機能が（遥かに複雑に）干渉することを取り上げる。例えば、パラメータ多相（第 26 章、第 28 章)、再帰型（第 20 章、第 21 章)、型演算子（第 31 章）などである。

型指定とキャスト

型指定演算子 t as T は、検査可能なドキュメント化機能の一種として 11.4 節で導入された。プログラマはこの機能を使って、ある複雑な式の部分項がある特定の型を持つというアサーションをプログラムのテキスト中に書ける。本書の例では、型指定を用いて、型を表示する方法を制御することもできる。項に対して実際に計算された型の代わりに、もっと読みやすい短縮された形式を型検査器に使わせるということである。

Java や C++ のように部分型付けを含む言語では、型指定はもっと興味深くなる。それらの言語では、型指定はしばしば**キャスト**と呼ばれ、(T)t と書かれる。キャストには二つのかなり異なる形式がある。いわゆる**アップキャスト**と**ダウンキャスト**である。アップキャストは単純であるが、ダウンキャストは**動的型テスト**を含み、大規模な拡張を必要とする。

アップキャストは、ある項に対して型検査器がその項に自然に割り当てる型の**上位型**を指定する、標準的な型指定演算子の具体例である。項 t と型 T を与え、t を T 型と「みなす」という意図を表す。型検査器は T が実際に、t が持ちうる型の一つであることを、以下の導出を構築することで検証する。

$$\dfrac{\dfrac{\dfrac{\vdots}{\Gamma \vdash \mathtt{t} : \mathtt{S}} \qquad \dfrac{\vdots}{\mathtt{S} <: \mathtt{T}}}{\Gamma \vdash \mathtt{t} : \mathtt{T}}\ \text{T-SUB}}{\Gamma \vdash \mathtt{t}\ \mathtt{as}\ \mathtt{T} : \mathtt{T}}\ \text{T-ASCRIBE}$$

これは t の「自然」な型付けと、包摂規則 T-SUB、そして次の型指定規則（11.4 節）を用いている。

$$\frac{\Gamma \vdash \mathtt{t}_1 : \mathtt{T}}{\Gamma \vdash \mathtt{t}_1\ \mathtt{as}\ \mathtt{T} : \mathtt{T}} \qquad \text{(T-ASCRIBE)}$$

アップキャストは**抽象化**の一種と考えられる。すなわち、値のうちの幾つかの部分の存在を隠して、周囲の文脈で使えなくしてしまう方法である。例えば t がレコード（あるいはもっと一般的にオブジェクト）であるとき、アップキャストを用いて一部のフィールド（メソッド）を隠してしまうことができる。

一方ダウンキャストでは、項に対して、型検査器が静的には導出できない型を割り当てられるようになる。ダウンキャストを許すには、as の型付け規則にいくらか驚くべき変更を行う。

$$\frac{\Gamma \vdash \mathtt{t}_1 : \mathtt{S}}{\Gamma \vdash \mathtt{t}_1\ \mathtt{as}\ \mathtt{T} : \mathtt{T}} \qquad \text{(T-DOWNCAST)}$$

つまり、$\mathtt{t}_1$ が正しく型付けされる（すなわち、ある型 S を持つ）ことを検査し、それから、S と T の間の関係について一切要求することなく、$\mathtt{t}_1$ に型 T を割り当てる。例えばダウンキャストを用い

[†3] この節で議論する拡張のほとんどは、fullsub 検査器では実装されていない。

ると、任意の引数を取り、数を格納するフィールド a を含むレコードへその引数を下向きにキャストし、そしてその数を返すような関数 f が書ける。

```
f = λx:Top. (x as {a:Nat}).a;
```

これは実質的に、プログラマが型検査器に「私は、(型付け規則の観点で説明するのはとても大変な諸般の事情で) f は**必ず**数の型のフィールド a を持つレコードの引数に適用されることを知っている。この件については私を信用してほしい」と伝えているといえる。

当然、そんな主張を盲目的に信じれば我々の言語の安全性に破滅的な影響がある。何らかの理由でプログラマがミスをして、フィールド a を持たないレコードに f を適用してしまったら、(コンパイラの内部次第で) どのような結果になってもおかしくない。そうではなく、我々のモットーは「信じるが、検証はする」であるべきだ。型検査器はコンパイル時に、ダウンキャストで指示された型を単純に受け入れる。しかしコンパイラは実行時の検査を挿入し、実際の値が本当に宣言された型を持っていることを検証するのである。つまり、型指定の元々の評価規則は注釈をただ捨てて

$$\mathsf{v}_1 \text{ as } \mathsf{T} \longrightarrow \mathsf{v}_1 \tag{E-Ascribe}$$

としていたが、そうすべきではなく、最初にその値の実際の (実行時の) 型を宣言された型と比較すべきなのである。

$$\frac{\vdash \mathsf{v}_1 : \mathsf{T}}{\mathsf{v}_1 \text{ as } \mathsf{T} \longrightarrow \mathsf{v}_1} \tag{E-Downcast}$$

例えば、f を引数 {a=5,b=true} に適用する場合、この規則は ⊢ {a=5,b=true} : {a:Nat} を検査する (この検査は成功する)。一方、f を {b=true} に適用する場合、E-Downcast の規則は適用されず、評価はこの時点で行き詰まり状態になる。この実行時の検査によって型保存の性質が取り戻される。

演習 15.5.1. [★★ ↛] これを証明せよ。 □

当然、進行定理は失われる。正しく型付けされたプログラムでも、誤ったダウンキャストを評価しようとすれば行き詰まり状態になってしまうからである。ダウンキャストを提供する言語は通常、この問題に対して二つのうちいずれかの方法で対処している。一つは、ダウンキャストが失敗した場合に動的な例外を送出し、プログラムが捕捉・対処できるようにする方法である (第 14 章を参照)。もう一つは、ダウンキャスト演算子を型の動的なテストの一種で置き換える方法である。

$$\frac{\Gamma \vdash \mathsf{t}_1 : \mathsf{S} \qquad \Gamma, \mathsf{x}{:}\mathsf{T} \vdash \mathsf{t}_2 : \mathsf{U} \qquad \Gamma \vdash \mathsf{t}_3 : \mathsf{U}}{\Gamma \vdash \text{if } \mathsf{t}_1 \text{ in } \mathsf{T} \text{ then } \mathsf{x} \to \mathsf{t}_2 \text{ else } \mathsf{t}_3 : \mathsf{U}} \tag{T-Typetest}$$

$$\frac{\vdash \mathsf{v}_1 : \mathsf{T}}{\text{if } \mathsf{v}_1 \text{ in } \mathsf{T} \text{ then } \mathsf{x} \to \mathsf{t}_2 \text{ else } \mathsf{t}_3 \longrightarrow [\mathsf{x} \mapsto \mathsf{v}_1]\mathsf{t}_2} \tag{E-Typetest1}$$

$$\frac{\nvdash \mathsf{v}_1 : \mathsf{T}}{\text{if } \mathsf{v}_1 \text{ in } \mathsf{T} \text{ then } \mathsf{x} \to \mathsf{t}_2 \text{ else } \mathsf{t}_3 \longrightarrow \mathsf{t}_3} \tag{E-Typetest2}$$

ダウンキャストは、Java のような言語では実際、とても日常的に用いられる。特に、ダウンキャストは一種の「貧者の多相性」を提供する。例えば Java では、Set や List のような「コレクショ

ンクラス」が単相的である[†4]。すべての型Tに対して型List T（型Tの要素を持つリスト）を提供する代わりに、Javaでは単にListを提供する。これは、最大の型Objectに属する要素からなるリストの型である。JavaにおいてObjectは、他のすべてのオブジェクト型の上位型であるから、リストにはどんなものでも格納できることになる。リストに要素を追加したいときは、単に包摂を用いてその型をObjectに昇格する。しかしリストから要素を**取り出す**とき、型検査器が知っているのは、その型がObject型であるということだけである。この型は、オブジェクトのほとんどのメソッド呼び出しを正当化しない。なぜならObject型は、すべてのJavaオブジェクトで共有される、表示などの、極めて一般的なメソッドをほんの少し知っているだけだからである。そのオブジェクトで何か意味のあることをするためには、期待している型Tにダウンキャストするところから始めなければならない。

Javaの型システムを真の多相性（第23章）で拡張するほうがよいことは、例えばPizza（Odersky and Wadler [344]）やGJ（Bracha, Odersky, Stoutamire, and Wadler [54]）、PolyJ（Myers, Bank, and Liskov [332]）、NextGen（Cartwright and Steele [91]）の設計者たちによって議論されてきた。こちらのほうがダウンキャストのイディオムよりも安全かつ効率的であり、実行時のテストが要らない。一方でそのような拡張を行うと、他の多くの言語機能や型システムと干渉し、既に大きな言語がさらに著しく複雑になってしまう（例えばIgarashi, Pierce, and Wadler [231; 233]を参照）。このことから、ダウンキャストのイディオムは安全性と複雑性の間の合理的・実用的な妥協案であるという見方もある。

ダウンキャストは、**リフレクション**に関するJavaの機能で重要な役割を果たしている。プログラマはリフレクションを用いて、Javaランタイムシステムにバイトコードファイルを動的にロードさせ、それに含まれるクラスのインスタンスを作らせることができる。明らかに、ロードされるクラスがどんなものかをこの時点で静的に予測することはできない（例えばバイトコードファイルは必要に応じてネット上から取得してくる可能性がある）。よって型検査器にできることは、新しく作られるインスタンスに最大の型Objectを割り当てることだけである。繰り返しになるが、何かしら意味のあることをするには、その新しいオブジェクトを期待している型Tにダウンキャストし、バイトコードファイルが提供しているクラスがこの型と合致しない場合に投げられる実行時例外を処理しなければならない。そうすれば、そのオブジェクトを型Tとして使えるようになる。

ダウンキャストの議論の最後に、実装について記述しておこう。我々が与えた規則から、ダウンキャストを言語に含めると、型検査器の機構を実行時システムに含めることになりそうである。さらに悪いことに、値は通常、実行時とコンパイラ内部とでは異なった表現になっている（特に関数は、バイトコードもしくはネイティブの機械語命令にコンパイルされている）。そのため、動的検査で必要になる、型を計算する型検査器を**別途**書く必要があるだろう。これを避けるため、実際の言語では、ダウンキャストは**型タグ**と併せて導入する。型タグは単一ワードのタグ（MLのデータ型構築子や11.10節のバリアントタグの方法に似ている）で、コンパイル時の型の実行時の「残骸」を表現している。実行時の部分型検査にはこれで十分である。第19章で、この機構の具体例を詳しく説明する。

バリアント

バリアント（11.10節参照）の部分型付け規則はレコードの場合とほとんど同じである。唯一異なるのは、幅規則S-VariantWidthが、部分型から上位型へ変換する際に、新しいバリアントをなくすのではなく**追加する**ことを許している点である。この直観は、タグ付きの式<l=t>がバリアン

†4（訳注）Java 5.0で多相的なコレクションクラスが導入された。

→ <> <: $\lambda_{<:}$（図 15-1）と単純なバリアントの規則（図 11-11）の拡張

新しい構文形式

t ::= ... 項：

$<l=t>$ （as はなし） タグ付け

新しい型付け規則 $\boxed{\Gamma \vdash t : T}$

$$\frac{\Gamma \vdash t_1 : T_1}{\Gamma \vdash <l_1=t_1> : <l_1{:}T_1>} \qquad \text{(T-VARIANT)}$$

新しい部分型付け規則 $\boxed{S <: T}$

$$<l_i{:}T_i\ ^{i\in 1..n}> \ <: \ <l_i{:}T_i\ ^{i\in 1..n+k}> \qquad \text{(S-VARIANTWIDTH)}$$

$$\frac{\text{各 } i \text{ に対して } S_i <: T_i}{<l_i{:}S_i\ ^{i\in 1..n}> \ <: \ <l_i{:}T_i\ ^{i\in 1..n}>} \qquad \text{(S-VARIANTDEPTH)}$$

$$\frac{<k_j{:}S_j\ ^{j\in 1..n}> \text{ は } <l_i{:}T_i\ ^{i\in 1..n}> \text{ の並べ替えである}}{<k_j{:}S_j\ ^{j\in 1..n}> \ <: \ <l_i{:}T_i\ ^{i\in 1..n}>} \qquad \text{(S-VARIANTPERM)}$$

▲ 図 15-5. バリアントと部分型付け

ト型 $<l_i{:}T_i\ ^{i\in 1..n}>$ に属しているのは、ラベル l が、型の中で列挙されているラベルの候補 $\{l_i\}$ のいずれかである場合だということである。したがって、この集合にラベルを追加すると、要素に関してこの集合から我々が読み取れる情報は減ってしまう。要素が一つしかないバリアント型 $<l_1{:}T_1>$ からはその要素が何のラベルでタグ付けされているかが正確にわかるし、二つのバリアントを持つ型 $<l_1{:}T_1, l_2{:}T_2>$ からはその要素がラベル l_1 またはラベル l_2 を持っていることがわかる。他も同様である。逆に、そのバリアントの値を**使用する**際には、それは必ず case 文の文脈の中にある。case 文は、型で列挙されているバリアントそれぞれに対して一つの分岐を持つはずである。バリアントを増やすと、case 文に対して不要な分岐を余計に追加しないといけないことになってしまう。

部分型付けとバリアントを組み合わせることの別の帰結として、タグ付け構文から注釈を外すことができる。11.10 節では $<l=t>$ as $<l_i{:}T_i\ ^{i\in 1..n}>$ と書いたが、今回は単に $<l=t>$ と書き、タグ付けの型付け規則を変えて $<l_1=t_1>$ に正確な型 $<l_1{:}T_1>$ を割り当てるようにする。より大きなバリアント型を得るには、包摂と S-VARIANTWIDTH を用いればよい。

リスト

これまでに、多くの共変の型構築子（レコード、バリアント、そして関数型の右側）と、一つの反変の型構築子（矢印型の左側）を見てきた。List 構築子もまた共変である。もし、型 S_1 の要素を持つリストがあり、かつ $S_1 <: T_1$ であれば、そのリストが型 T_1 の要素を持っているとみなしても安全である。

$$\frac{S_1 <: T_1}{\text{List } S_1 <: \text{List } T_1} \qquad \text{(S-LIST)}$$

参照

すべての型構築子が共変か反変であるというわけではない。例えば Ref 構築子は、型安全性を保つためには**非変**でなければならない。

$$\frac{S_1 <: T_1 \qquad T_1 <: S_1}{\text{Ref } S_1 <: \text{Ref } T_1} \qquad \text{(S-REF)}$$

Ref S_1 が Ref T_1 の部分型であるためには、S_1 と T_1 が部分型関係の下で**同等**であること（つまり、お互いに部分型であること）を要請する。これにより、Ref 構築子の下でレコードのフィールドを並べ替える自由が得られる（例えば、Ref {a:Bool,b:Nat} <: Ref {b:Nat,a:Bool}）。しかし、それ以上のものではない。

この部分型付け規則がこのようにとても制限されているのは、型 Ref T_1 の値がその文脈の中で、異なる二種類の方法で使われうるからである。すなわち、読み出し（!）と書き込み（:=）である。読み出しの場合、文脈は型 T_1 の値を得ることを期待するため、もし参照が実際には型 S_1 の値を生成するなら、文脈の期待を裏切らないよう、S_1 <: T_1 が必要である。一方、同じ参照セルが書き込みに使われるならば、文脈から与えられる新しい値は型 T_1 を持つだろう。もし参照の実際の型が Ref S_1 であれば、他の誰かが後にこの値を読み出し、型 S_1 の値として使う可能性がある。これは、T_1 <: S_1 が成立する場合のみ安全である。

演習 15.5.2. [★ ↛]　(1) S-Ref の一つ目の前提をなくした場合に実行時型エラーで失敗する（すなわち、評価が行き詰まり状態になる）短いプログラムを書け。(2) 二つ目の前提をなくした場合に失敗するプログラムを書け。 □

配列

参照の部分型付け規則を非変にすることになった要因は、配列にも明らかに当てはまる。配列は操作として取り出しと破壊的代入の両方を含むからである。

$$\frac{S_1 <: T_1 \qquad T_1 <: S_1}{\texttt{Array}\ S_1 <: \texttt{Array}\ T_1} \qquad \text{(S-Array)}$$

面白いことに、Java は実際には配列に関して共変な部分型付けを許している。

$$\frac{S_1 <: T_1}{\texttt{Array}\ S_1 <: \texttt{Array}\ T_1} \qquad \text{(S-ArrayJava)}$$

（ただし、Java の文法では S_1[] <: T_1[]。）元々この機能が導入されたのは、配列の一部分を複製するなどの一部の基本的な操作の型付けにおいて、パラメータ多相がないことを補うためであった。しかし現在、一般的には、これは言語設計の欠陥であると考えられている。これは、配列を扱うプログラムの性能に重大な影響を与えるためである。その理由は、健全でない部分型付け規則のせいで、配列に代入される値が配列の要素の実際の型（の部分型）であることを保証するため、**あらゆる**配列への**すべての**破壊的代入において実行時検査をする必要があるからである。

演習 15.5.3. [★★★ ↛]　型検査が成功するにもかかわらず、実行時に（ArrayStoreException を送出して）失敗する、配列を用いた短いプログラムを書け。 □

参照、再び

参照に関するより洗練された分析は、二つの新しい型構築子 Source と Sink を導入することで得られる。これは最初に Reynolds によって Forsythe 言語 [415] で探求された。Source T は直観的には、型 T の値をセルから読み出せる能力だと考えられる（しかし破壊的代入は許していない）。一方 Sink は、セルへの書き込み能力である。Ref T はこの二つの能力の組み合わせであり、読み書きの両方を許す。

参照先の値の取り出し (dereferencing) と破壊的代入 (assignment) に対する型付け規則 (図 13-1)

は、適切な能力だけを要求するように変更される。

$$\frac{\Gamma \mid \Sigma \vdash \mathtt{t}_1 : \mathtt{Source}\ \mathtt{T}_{11}}{\Gamma \mid \Sigma \vdash \mathtt{!t}_1 : \mathtt{T}_{11}} \qquad \text{(T-Deref)}$$

$$\frac{\Gamma \mid \Sigma \vdash \mathtt{t}_1 : \mathtt{Sink}\ \mathtt{T}_{11} \qquad \Gamma \mid \Sigma \vdash \mathtt{t}_2 : \mathtt{T}_{11}}{\Gamma \mid \Sigma \vdash \mathtt{t}_1\mathtt{:=}\mathtt{t}_2 : \mathtt{Unit}} \qquad \text{(T-Assign)}$$

ここで、セルから値を読み出す能力だけを持ち、かつ、これらの値が型 $\mathtt{S}_1$ を持っていることが保証されているとすると、$\mathtt{S}_1$ が $\mathtt{T}_1$ の部分型である限り、この能力を型 $\mathtt{T}_1$ の値を読み出す能力に「格下げ」しても安全である。すなわち、Source 構築子は共変である。

$$\frac{\mathtt{S}_1 <: \mathtt{T}_1}{\mathtt{Source}\ \mathtt{S}_1 <: \mathtt{Source}\ \mathtt{T}_1} \qquad \text{(S-Source)}$$

逆に、あるセルに型 $\mathtt{S}_1$ の値を書き込む能力は、より**小さい**型 $\mathtt{T}_1$ を書き込む能力に格下げできる。すなわち、Sink 構築子は反変である。

$$\frac{\mathtt{T}_1 <: \mathtt{S}_1}{\mathtt{Sink}\ \mathtt{S}_1 <: \mathtt{Sink}\ \mathtt{T}_1} \qquad \text{(S-Sink)}$$

最後に、Ref $\mathtt{T}_1$ が読み書き能力の組み合わせであるという直観は、Ref を Source か Sink へ格下げできるような二つの部分型付け規則によって表現できる。

$$\mathtt{Ref}\ \mathtt{T}_1 <: \mathtt{Source}\ \mathtt{T}_1 \qquad \text{(S-RefSource)}$$

$$\mathtt{Ref}\ \mathtt{T}_1 <: \mathtt{Sink}\ \mathtt{T}_1 \qquad \text{(S-RefSink)}$$

チャネル

同じ直観（と同じ部分型付け規則）は、最近の並行プログラミング言語（例えば Pict（Pierce and Turner [376]; Pierce and Sangiorgi [370]））において、**チャネル型**の扱いの基礎をなしている。重要な観察は、型付けの観点で通信チャネルは参照セルと全く同じように振る舞い、参照と同じように読み書きできることである。さらに、どの読み込みがどの書き込みに対応するか静的に判断するのは難しいので、型安全性を保証する唯一の単純な方法は、チャネルで受け渡しされるすべての値が同じ型に属さなければならないとすることである。仮に、あるチャネルへの書き込み能力だけを誰かに渡したとする。その人がこの能力を、より小さい型の値を書き込むと約束する他の人に渡しても安全である。つまり、「出力チャネル」の型構築子は反変である。同様に、あるチャネルへの読み込み能力だけを渡したとして、その能力をより大きい型の値を読み出す能力に格下げしても安全である。つまり、「入力チャネル」構築子は共変である。

基本型

豊富な基本型集合を持つ本格的な言語では、基本型の間にプリミティブな部分型関係を導入するとしばしば便利である。例えば、多くの言語ではブール値 true と false を、実際には数 1 と 0 として表現している。もし望めば、部分型付けの公理である Bool <: Nat を導入することによって、プログラマにこの事実を露出できる。これで、if b then 5 else 0 の代わりに 5*b のようなコンパクトな表現を書けるようになる。

15.6 部分型付けに対する型強制意味論

本章におけるここまでの我々の直観は、部分型は「意味論的に重要でない」というものであった。部分型があってもプログラムの評価方法は変わることはなく、むしろ部分型は項の型付けの柔軟性を増やすためだけの手段であった。この解釈は単純かつ自然であるが、性能面ではいくらか不利になる。特に、数値計算とレコードのフィールドアクセスで不利になり、これは高性能な実装では許容できるものではない。本節では代替となる**型強制意味論**の概略を説明し、その場合に発生する幾つかの新たな問題について議論する。本節は望むなら読み飛ばしてもかまわない。

部分集合意味論の問題

15.5 節で見たように、異なる基本型の間で部分型付けを許すと便利になる場合がある。しかし、ある基本型が別の基本型を含むことが「直観的には合理的」でも、性能上に悪影響を与えることがある。例えば仮に、浮動小数点演算の中で明示的な型強制なしで整数を書けるよう、`Int <: Float` という公理を導入したとする。これにより、例えば `4.5 + intToFloat(6)` と書いていたところを `4.5 + 6` と書けるようになる。この部分集合意味論の下では、文字通り、整数値の集合が浮動小数点数の集合の部分集合でなければならないことを意味する。しかしほとんどの実際の計算機では、整数と浮動小数点数の具体的な表現は全く異なる。整数は通常 2 の補数形式で表されるが、一方、浮動小数点数は仮数、指数、符号に分けられ、さらに NaN（非数）のような特別な場合もある。

これらの表現の違いと、部分型付けの部分集合意味論を両立させるためには、**タグ付き**の（または、**ボックス化**された）共通の表現を数に用いればよい。整数は、計算機上の整数にタグを付けたものとして表す（このタグは、別のヘッダワードの中か、実際の整数と同じワードの高位ビットの中に付ける）。浮動小数点数は、計算機上の浮動小数点数に異なるタグを付けたものとして表す。型 `Float` はタグ付きの値全体の集合（浮動小数点数と整数の両方）を指し、`Int` はタグ付きの整数のみを指す。

このやり方は不合理というわけではなく、現代的な言語実装の多くが実際に用いている表現手段と対応しており、そうした言語実装ではガベージコレクションをサポートするためにもタグビット（もしくはワード）が必要になる。不都合な点は、数に対するプリミティブの演算すべての実装が、引数のタグ検査、プリミティブの数をアンボックスする数個の命令、実際の演算を行う 1 命令、結果を再びボックス化する数個の命令、という命令列になってしまうことである。賢いコンパイラであれば、最適化によってオーバーヘッドをある程度取り除けるが、しかし現在利用可能な最高の技術をもってしても、性能は大幅に悪化してしまう。グラフィックや科学計算のように重たい数値演算のコードでは特に顕著になる。

レコードと部分型付けを組み合わせると（特に、並べ替え規則によって）、別の性能上の問題が発生する。フィールド射影のための単純な評価規則

$$\{\mathtt{l}_i\mathtt{=v}_i{}^{i\in 1..n}\}\mathtt{.l}_j \longrightarrow \mathtt{v}_j \qquad \text{(E-ProjRcd)}$$

は、「レコードのラベルから $\mathtt{l}_j$ を探し、関連付けられた値 $\mathtt{v}_j$ を返す」と読める。しかし実際の実装では、望むラベルを見つけるのにレコードのフィールド全部を実行時に線形探索するようなことは、明らかにやりたくない。部分型付けがない言語では（もしくは部分型付けがあるが並べ替え規則がない言語では）、もっとうまいやり方がある。ラベル $\mathtt{l}_j$ がレコードの**型**の三番目に現れるなら、この型を持つ実行時の値はすべて $\mathtt{l}_j$ を三番目のフィールドとして持つとわかっているので、実行時にラベルを調べる必要は全くない（実際、実行時の表現からラベルを完全に取り除いてしまえる。事実上、レコードを組に翻訳していることになる）。フィールド $\mathtt{l}_j$ の値を得るには、レコードの先頭

を指すレジスタに対して 3 ワードの定数オフセットを付けてアクセスする間接読み込み命令を生成する。この技法は、並べ替え規則があると不可能になる。というのも、あるレコードの値が、$\mathtt{l}_j$ が三番目のフィールドにある型に属すると知っていたところで、フィールド $\mathtt{l}_j$ が実際にレコードのどこに格納されているかについては何もわからないからである。これも、賢い最適化と実行時の工夫で問題を軽減できるが、一般的には、フィールドの射影には何らかの形で実行時の探索が必要になりうる[†5]。

型強制意味論

以上二つの問題は、異なる意味論を導入することで対処できる。それは、部分型付けを実行時の型強制に置き換えることで「コンパイル時に消去」してしまうことである。例えば、型検査の際に Int が Float に昇格するなら、この数の表現を計算機上の整数から計算機上の浮動小数点数へと実行時に物理的に変換する。同様に、レコードの並べ替えの部分型付け規則を使っているところは、フィールドの順序をその通りに再配置するコード断片にコンパイルされるようにする。これで、プリミティブな数の演算やフィールドアクセスをアンボックス化や検索のオーバーヘッドなしで実行できるようになる。

直観的には、部分型付けを持つ言語の型強制意味論は、この言語から、部分型付けを含まない低水準な言語へ項を変換する関数として表現される。究極的には、低水準言語は具体的なプロセッサの機械語かもしれない。しかし説明が目的なら、議論を抽象的な段階に保てる。変換元の言語としては、本章のほとんどで用いてきた、単純型付きラムダ計算に部分型付けとレコードを持たせたものを用いる。変換先の低水準言語としては、純粋な単純型付きラムダ計算にレコードと Unit 型を持たせたものを用いる（Unit 型は Top を解釈するために用いる）。

形式的には、コンパイルは三つの翻訳関数からなる。それぞれ、型の翻訳、部分型付けの翻訳、型付けの翻訳である。型の翻訳に関しては、単に Top を Unit に置き換えるだけである。この関数を $[\![-]\!]$ と書く。

$$
\begin{aligned}
[\![\mathsf{Top}]\!] &= \mathtt{Unit} \\
[\![\mathsf{T}_1\to\mathsf{T}_2]\!] &= [\![\mathsf{T}_1]\!]\to[\![\mathsf{T}_2]\!] \\
[\![\{\mathtt{l}_i:\mathsf{T}_i\ ^{i\in 1..n}\}]\!] &= \{\mathtt{l}_i:[\![\mathsf{T}_i]\!]\ ^{i\in 1..n}\}
\end{aligned}
$$

例えば、$[\![\mathsf{Top}\to\{\mathtt{a}:\mathsf{Top},\mathtt{b}:\mathsf{Top}\}]\!] = \mathtt{Unit}\to\{\mathtt{a}:\mathtt{Unit},\mathtt{b}:\mathtt{Unit}\}$ である。（他の翻訳も $[\![-]\!]$ で書く。何について話しているのかは文脈から常に明らかである。）

項を翻訳するためには、その項の型検査で包摂が用いられた場所を知っている必要がある。そこが実行時の型強制を挿入する場所だからである。この観察を定式化するには、型付け判断式の**導出**の上の関数として翻訳を定義するのが手軽である。同様に、型 S の値を型 T に変換する型強制関数を生成するためには、S が T の部分型であるという**事実**だけでなく、**理由**まで知っている必要がある。これは、部分型付け導出から型強制を生成することで達成できる。

翻訳を定式化するために、導出に名前を付ける簡潔な記法が必要である。$\mathcal{C} :: \mathsf{S} <: \mathsf{T}$ と書いたら、「$\mathcal{C}$ が部分型付け導出木で、その結論が $\mathsf{S} <: \mathsf{T}$」を意味するものとする。同様に、$\mathcal{D} :: \Gamma \vdash \mathtt{t} : \mathsf{T}$ と書いたら、「$\mathcal{D}$ が型付け導出で、その結論が $\Gamma \vdash \mathtt{t} : \mathsf{T}$」を意味する。

まずは、部分型付け判断式 $\mathsf{S} <: \mathsf{T}$ の導出 $\mathcal{C}$ が与えられたときに型強制 $[\![\mathcal{C}]\!]$ を生成する関数を見

[†5] オブジェクトの部分型付けが並べ替えを許している言語では、オブジェクトのフィールドやメソッドにアクセスする場合でも同様の現象が見られる。このために、例えば Java は、部分型付けをクラス間だけに制限している。そうすることで、新しいフィールドが追加できる位置を最後だけにしているのである。インターフェイス同士の間（およびインターフェイスとクラスとの間）の部分型付けは、並べ替えを許している（そうしなければ、インターフェイスはほとんど何の役にも立たないものになる）。マニュアルでも明確に、インターフェイスからメソッドを探すことはクラスから探すよりも一般的に遅くなると警告されている。

てみよう。この型強制は単に、型 $[\![\mathtt{S}]\!]$ から型 $[\![\mathtt{T}]\!]$ への（翻訳先の言語 $\lambda_{\rightarrow}$ で書かれた）関数である。この定義は C で用いられる最後の規則の場合分けに基づいている。

$$\left[\!\!\left[\frac{}{\mathtt{T} <: \mathtt{T}}\ (\text{S-Refl})\right]\!\!\right] = \lambda\mathtt{x}{:}[\![\mathtt{T}]\!].\ \mathtt{x}$$

$$\left[\!\!\left[\frac{}{\mathtt{S} <: \mathtt{Top}}\ (\text{S-Top})\right]\!\!\right] = \lambda\mathtt{x}{:}[\![\mathtt{S}]\!].\ \mathtt{unit}$$

$$\left[\!\!\left[\frac{C_1 :: \mathtt{S} <: \mathtt{U} \qquad C_2 :: \mathtt{U} <: \mathtt{T}}{\mathtt{S} <: \mathtt{T}}\ (\text{S-Trans})\right]\!\!\right] = \lambda\mathtt{x}{:}[\![\mathtt{S}]\!].\ [\![C_2]\!]([\![C_1]\!]\ \mathtt{x})$$

$$\left[\!\!\left[\frac{C_1 :: \mathtt{T}_1 <: \mathtt{S}_1 \qquad C_2 :: \mathtt{S}_2 <: \mathtt{T}_2}{\mathtt{S}_1\rightarrow\mathtt{S}_2 <: \mathtt{T}_1\rightarrow\mathtt{T}_2}\ (\text{S-Arrow})\right]\!\!\right] = \lambda\mathtt{f}{:}[\![\mathtt{S}_1\rightarrow\mathtt{S}_2]\!].\ \lambda\mathtt{x}{:}[\![\mathtt{T}_1]\!].\ [\![C_2]\!](\mathtt{f}([\![C_1]\!]\ \mathtt{x}))$$

$$\left[\!\!\left[\frac{}{\{\mathtt{l}_i{:}\mathtt{T}_i\ ^{i\in 1..n+k}\} <: \{\mathtt{l}_i{:}\mathtt{T}_i\ ^{i\in 1..n}\}}\ (\text{S-RcdWidth})\right]\!\!\right] = \lambda\mathtt{r}{:}\{\mathtt{l}_i{:}[\![\mathtt{T}_i]\!]\ ^{i\in 1..n+k}\}.\ \{\mathtt{l}_i\mathtt{=r.l}_i\ ^{i\in 1..n}\}$$

$$\left[\!\!\left[\frac{\text{各 } i \text{ に対して } C_i :: \mathtt{S}_i <: \mathtt{T}_i}{\{\mathtt{l}_i{:}\mathtt{S}_i\ ^{i\in 1..n}\} <: \{\mathtt{l}_i{:}\mathtt{T}_i\ ^{i\in 1..n}\}}\ (\text{S-RcdDepth})\right]\!\!\right] = \lambda\mathtt{r}{:}\{\mathtt{l}_i{:}[\![\mathtt{S}_i]\!]\ ^{i\in 1..n}\}.\ \{\mathtt{l}_i\mathtt{=}[\![C_i]\!](\mathtt{r.l}_i)\ ^{i\in 1..n}\}$$

$$\left[\!\!\left[\frac{\{\mathtt{k}_j{:}\mathtt{S}_j\ ^{j\in 1..n}\} \text{ は } \{\mathtt{l}_i{:}\mathtt{T}_i\ ^{i\in 1..n}\} \text{ の並べ替えである}}{\{\mathtt{k}_j{:}\mathtt{S}_j\ ^{j\in 1..n}\} <: \{\mathtt{l}_i{:}\mathtt{T}_i\ ^{i\in 1..n}\}}\ (\text{S-RcdPerm})\right]\!\!\right] = \lambda\mathtt{r}{:}\{\mathtt{k}_j{:}[\![\mathtt{S}_j]\!]\ ^{j\in 1..n}\}.\ \{\mathtt{l}_i\mathtt{=r.l}_i\ ^{i\in 1..n}\}$$

補題 15.6.1. $C :: \mathtt{S} <: \mathtt{T}$ ならば、$\vdash [\![C]\!] : [\![\mathtt{S}]\!]\rightarrow[\![\mathtt{T}]\!]$ □

［証明］ C に関する単純な帰納法。 □

型付け導出は同様に変換される。$\mathcal{D}$ が型付け判断式 $\Gamma \vdash \mathtt{t} : \mathtt{T}$ の導出ならば、その翻訳 $[\![\mathcal{D}]\!]$ は翻訳先の言語の項で、型 $[\![\mathtt{T}]\!]$ を持つ。この翻訳関数はしばしば **Penn 翻訳**と呼ばれる。最初にこれを研究した Pennsylvania 大学のグループにちなんでいる（Breazu-Tannen, Coquand, Gunter, and Scedrov [57]）。

$$\left[\!\!\left[\frac{\mathtt{x}{:}\mathtt{T} \in \Gamma}{\Gamma \vdash \mathtt{x} : \mathtt{T}}\ (\text{T-Var})\right]\!\!\right] = \mathtt{x}$$

$$\left[\!\!\left[\frac{\mathcal{D}_2 :: \Gamma, \mathtt{x}{:}\mathtt{T}_1 \vdash \mathtt{t}_2 : \mathtt{T}_2}{\Gamma \vdash \lambda\mathtt{x}{:}\mathtt{T}_1.\mathtt{t}_2 : \mathtt{T}_1\rightarrow\mathtt{T}_2}\ (\text{T-Abs})\right]\!\!\right] = \lambda\mathtt{x}{:}[\![\mathtt{T}_1]\!].\ [\![\mathcal{D}_2]\!]$$

$$\left[\!\!\left[\frac{\mathcal{D}_1 :: \Gamma \vdash \mathtt{t}_1 : \mathtt{T}_{11}\rightarrow\mathtt{T}_{12} \qquad \mathcal{D}_2 :: \Gamma \vdash \mathtt{t}_2 : \mathtt{T}_{11}}{\Gamma \vdash \mathtt{t}_1\ \mathtt{t}_2 : \mathtt{T}_{12}}\ (\text{T-App})\right]\!\!\right] = [\![\mathcal{D}_1]\!]\ [\![\mathcal{D}_2]\!]$$

$$\left[\!\!\left[\frac{\text{各 } i \text{ に対して } \mathcal{D}_i :: \Gamma \vdash \mathtt{t}_i : \mathtt{T}_i}{\Gamma \vdash \{\mathtt{l}_i\mathtt{=t}_i\ ^{i\in 1..n}\} : \{\mathtt{l}_i{:}\mathtt{T}_i\ ^{i\in 1..n}\}}\ (\text{T-Rcd})\right]\!\!\right] = \{\mathtt{l}_i\mathtt{=}[\![\mathcal{D}_i]\!]\ ^{i\in 1..n}\}$$

$$\left[\!\!\left[\frac{\mathcal{D}_1 :: \Gamma \vdash \mathtt{t}_1 : \{\mathtt{l}_i{:}\mathtt{T}_i\ ^{i\in 1..n}\}}{\Gamma \vdash \mathtt{t}_1.\mathtt{l}_j : \mathtt{T}_j}\ (\text{T-Proj})\right]\!\!\right] = [\![\mathcal{D}_1]\!].\mathtt{l}_j$$

$$\left[\!\!\left[\frac{\mathcal{D} :: \Gamma \vdash \mathtt{t} : \mathtt{S} \qquad C :: \mathtt{S} <: \mathtt{T}}{\Gamma \vdash \mathtt{t} : \mathtt{T}}\ (\text{T-Sub})\right]\!\!\right] = [\![C]\!]\ [\![\mathcal{D}]\!]$$

定理 15.6.2. $\mathcal{D} :: \Gamma \vdash \mathtt{t} : \mathtt{T}$ ならば $[\![\Gamma]\!] \vdash [\![\mathcal{D}]\!] : [\![\mathtt{T}]\!]$。ただし $[\![\Gamma]\!]$ は文脈に含まれる型それぞれを翻訳する拡張である。すなわち、$[\![\emptyset]\!] = \emptyset$ であり、$[\![\Gamma, \mathtt{x}{:}\mathtt{T}]\!] = [\![\Gamma]\!], \mathtt{x}{:}[\![\mathtt{T}]\!]$ である。 □

［証明］ $\mathcal{D}$ に関する単純な帰納法。補題 15.6.1 を T-Sub の場合に用いる。 □

このような評価規則を定義すると、部分型付けを持つ高水準言語に評価規則は不要になる。項を評価するには、評価規則の代わりに、(高水準言語の型付けと部分型付け規則を用いて) 型検査を行い、その型付け導出を低水準な言語に翻訳し、それからこの言語の評価関係を用いて操作的な振る舞いを得ればよい。この戦略は、部分型付けを持つ高性能な言語実装の幾つかで実際に使われている。例えばYaleコンパイラグループの実験的なJavaコンパイラ (League, Shao, and Trifonov [274]; League, Trifonov, and Shao [275]) がそうである。

演習 15.6.3. [★★★ ↛] (レコードの代わりに) **組**を持つ単純型付きラムダ計算を翻訳先言語として用いるよう、上記の翻訳を修正せよ。それでも定理 15.6.2 が成り立つことを検査せよ。 □

整合性

部分型付けを持つ言語に型強制意味論を与えることには、慎重に回避する必要のある落とし穴が潜んでいる。例えば仮に、基本型 `Int`、`Bool`、`Float`、`String` で今の言語を拡張するとしよう。以下のプリミティブの型強制はすべて有用かもしれない。

$$
\begin{aligned}
[\![\texttt{Bool} <: \texttt{Int}]\!] &= \lambda\texttt{b:Bool. if b then 1 else 0} \\
[\![\texttt{Int} <: \texttt{String}]\!] &= \texttt{intToString} \\
[\![\texttt{Bool} <: \texttt{Float}]\!] &= \lambda\texttt{b:Bool. if b then 1.0 else 0.0} \\
[\![\texttt{Float} <: \texttt{String}]\!] &= \texttt{floatToString}
\end{aligned}
$$

関数 `intToString` や `floatToString` は数の文字列表現を構築するプリミティブである。この例では、`intToString(1) = "1"` とし、`floatToString(1.0) = "1.000"` としよう。

ここで、型強制意味論を用いて次の項を評価するとする。

```
(λx:String.x) true;
```

この項は、プリミティブな型に対する上記の公理が与えられれば、型付け可能である。事実、二つの異なる方法でこれを型付けできる。型 `String→String` の関数の引数として `true` が適切であることを示すには、包摂を用いて `Bool` を `Int` へ変換し、それから `String` に変換するか、または、`Bool` を `Float` へ変換し、それから `String` に変換すればよい。しかし、この二つの導出を $\lambda_{\rightarrow}$ へ変換すると、異なる振る舞いになってしまう。`true` を型 `Int` に型強制すると 1 を得るので、`intToString` は文字列 `"1"` を返す。しかしもし `true` を浮動小数に型強制し、それから `floatToString` で `String` にした場合 (つまり、`true : String` が `Float` を経由して証明されている型付け導出の構造に従った場合)、`"1.000"` を得ることになってしまう。しかし、`"1"` と `"1.000"` は全く異なる文字列である。長さすら違う。言い換えれば、⊢ `(λx:String. x) true : String` の証明方法の選び方が、翻訳されたプログラムの振る舞いに影響を与えるということなのだ。しかし、この選び方は完全にコンパイラ内部の話である (プログラマは単に**項**を書くだけであり、導出を書くわけではない)。よって、プログラマが自分で書いたプログラムの振る舞いについて制御も予想もできないような言語を設計してしまったことになる。

このような問題に対する正しい対処は、**整合性**と呼ばれる、翻訳関数の定義に対する追加の要求を課すことである。

定義 15.6.4. ある言語の型付け導出から他の言語の項への翻訳 $[\![-]\!]$ が**整合性を持つ**とは、$\Gamma \vdash$ `t : T` を結論に持つ任意の導出の組 $\mathcal{D}_1$ と $\mathcal{D}_2$ に対して、翻訳 $[\![\mathcal{D}_1]\!]$ と $[\![\mathcal{D}_2]\!]$ が翻訳先の言語の項として振る舞い等価になることをいう。 □

具体的には、上記で与えた（基本型のない）翻訳は整合性を持つ。（上記の公理を持つ）基本型を考えているときでも整合性を取り戻すには、`floatToString(0.0) = "0"` や `floatToString(1.0) = "1"` となるように `floatToString` プリミティブの定義を変更すれば十分である。

整合性を証明することは、複雑な言語においては特に、ややこしい問題となりうる。Reynolds [411]、Breazu-Tannen et al. [57]、Curien and Ghelli [127]、および Reynolds [418] を見よ。

15.7 交差型と合併型

部分型関係の強力な改良は、型の言語に**交差**（intersection）演算子を追加することで得られる。交差型は、Coppo、Dezani、Salleé、Pottinger が発明した（Coppo and Dezani-Ciancaglini [115]; Coppo, Dezani-Ciancaglini, and Sallé [116]; Pottinger [392]）。わかりやすい概論は、Reynolds [415; 420]、Hindley [208]、Pierce [365] に見られる。

交差型 $T_1 \wedge T_2$ の要素は、T_1 と T_2 の両方に属する項である。すなわち、$T_1 \wedge T_2$ は、順序理論でいう T_1 と T_2 の交わり（下限）である。この直観は、三つの新しい部分型付け規則によって表現される。

$$T_1 \wedge T_2 <: T_1 \qquad \text{(S-INTER1)}$$

$$T_1 \wedge T_2 <: T_2 \qquad \text{(S-INTER2)}$$

$$\frac{S <: T_1 \qquad S <: T_2}{S <: T_1 \wedge T_2} \qquad \text{(S-INTER3)}$$

もう一つ、交差型と矢印型の自然な組み合わせを許す規則を加える。

$$(S \to T_1) \wedge (S \to T_2) <: S \to (T_1 \wedge T_2) \qquad \text{(S-INTER4)}$$

この規則の裏にある直観は、もし項が関数型 $S \to T_1$ と $S \to T_2$ を持つとわかっているならば、それに S を渡すと T_1 かつ T_2 の両方に属する値が返ってくると確実に期待できることである。

交差型の強力さは、以下の事実によって表現される。部分型付けと交差型を持つ単純型付きラムダ計算の名前呼びの変種では、型の割り当てが可能な型無しラムダ項の集合は**正規化可能**な項の集合と一致する。つまり、項が型付け可能なとき、かつそのときに限り、評価が停止する。これは直ちに、交差型を持つ体系の型再構築問題（第 22 章を見よ）が決定不能であることを意味する。

より実用的な観点で面白いのは、交差型がある種の**有限オーバーロード**を提供していることである。例えば、自然数の足し算としても浮動小数点数の足し算としても使える加算演算子に、型 $(\texttt{Nat}\to\texttt{Nat}\to\texttt{Nat}) \wedge (\texttt{Float}\to\texttt{Float}\to\texttt{Float})$ を割り当てられる（正しい命令を選択するには、例えば、引数の実行時の表現に含まれるタグビットを用いることができる）。

残念ながら交差型の強力さは、言語設計者にとって幾つか実用上の難しい問題を引き起こしてしまう。これまで、本格的な言語ではただ一つ、Forsythe（Reynolds [415]）だけが最も一般的な形で交差型を含んでいる。**詳細型**として知られる制限された形式のほうが扱いやすいといえるかもしれない（Freeman and Pfenning [160]; Pfenning [358]; Davies [132]）。

双対な概念である**合併型** $T_1 \vee T_2$ もまた、大変便利であるとわかる。和型やバリアント型と異なり（紛らわしいが、これらもときどき「合併型」と呼ばれる）、$T_1 \vee T_2$ が表すのは、T_1 に属する値の集合と T_2 に属する値の集合の、通常の合併（和集合）である。ある要素の出自を示すタグは付与されない。それゆえ、`Nat` ∨ `Nat` は実際には単に `Nat` の別の呼び方である。互いに素でない合併

型はプログラム解析において重要な役割を長い間果たしてきた（Palsberg and Pavlopoulou [349]）が、それを機能として持つプログラミング言語はほとんどない（特筆すべきは Algol 68 である。van Wijngaarden et al. [476] を参照せよ）。しかし近年、XML（Buneman and Pierce [67]; Hosoya, Vouillon, and Pierce [222]）のように「半構造化」されたデータベースのフォーマットを処理するための型システムの文脈で徐々に適用されている。

互いに素な合併型（直和型）と互いに素でない合併型の形式上の主な違いは、後者がどんな種類の case 構文も持たないことである。値 v が型 $T_1 \vee T_2$ を持つことだけがわかっているとき、v に対して安全に行える操作は、T_1 と T_2 の**両方**にとって意味をなす操作だけである（例えば T_1 と T_2 が共にレコードなら、共通のフィールドで v を射影することは意味をなす）。C におけるタグのない union 型は、型安全性を壊す原因である。これはまさに上記の制限を無視して、$T_1 \vee T_2$ の要素に対し T_1 か T_2 の**片方**でしか意味をなさない操作を許してしまっているからである。

15.8 注記

プログラミング言語における部分型付けのアイディアは 1960 年代まで遡り、Simula（Birtwistle, Dahl, Myhrhaug, and Nygaard [46]）とその親戚に見られる。最初に形式的に扱ったのは Reynolds [411] と Cardelli [73] である。

レコードを扱う型付け規則と（特に）部分型付け規則は、これまで見てきた他の規則と比べていくぶん重苦しい。可変個の前提（フィールドごとに一個）や、フィールドのインデックスの並べ替えなどの追加の機構を含むことになる。これらの規則を書く方法は他にも数多くあるが、いずれも同様の複雑さを抱えているか、非形式的な慣習（例えば省略記号「l_1:T_1 ... l_n:T_n」）を導入して避けている。この状況に不満を持った Cardelli と Mitchell は、**レコード演算**の計算体系を開発した [87]。そこでは、複数のフィールドを持つレコードを作るマクロな演算は、基本的な空のレコード値と、一度に一つのフィールドを追加する演算に分解される。フィールドの破壊的な更新やレコードの結合といった他の操作もこの枠組みで考えられる。これらの演算に対する型付け規則は、特にパラメータ多相があるとき、かなり微妙なものになる。そのため、ほとんどの言語設計者は普通のレコードにとどまることを好む。それでも、Cardelli と Mitchell の体系は重要な概念的成果として認められている。**列変数多相**に基づいてレコードを扱う別の方法もあり、これは Wand [487; 488; 490]、Rémy [400; 399; 403] などによって開発され、OCaml（Rémy and Vouillon [406]; Vouillon [477]）のオブジェクト指向の機能の基礎をなしている。

> 型理論が解決しようとする根本的な問題は、プログラムに意味があることを保証することだ。型理論によって引き起こされる根本的な問題は、意味のあるプログラムが指定された意味を持たないかもしれないことだ。この葛藤から、より強力な型システムを探し求める冒険が始まるのである。
>
> —*Mark Manasse*

第16章

部分型付けのメタ理論

本章で扱う体系は部分型付け（図 15-1）とレコード（図 15-3）を持つ単純型付きラムダ計算である。対応する OCaml 実装は `rcdsub` である。16.3 節では、ブール値と条件式（図 8-1）も取り扱う。その OCaml 実装は `joinsub` である。16.4 節では、`Bot` へ議論を拡張する。それに対応する実装は `bot` である。

前章での、部分型付けのある単純型付きラムダ計算の定義は、直接実装するには適していない。これまで見てきた他の計算体系とは違って、この型システムの規則は**構文主導**ではない。つまり単に「下から上に読む」だけでは型検査アルゴリズムは得られない。その主な原因は型付け関係における包摂規則（T-Sub）と部分型関係における推移律の規則（S-Trans）である。

T-Sub に問題がある理由は、結論にある項が裸のメタ変数 `t` で指定されていることにある。

$$\frac{\Gamma \vdash \mathtt{t} : \mathtt{S} \qquad \mathtt{S} <: \mathtt{T}}{\Gamma \vdash \mathtt{t} : \mathtt{T}} \qquad \text{(T-Sub)}$$

例えば T-Abs はラムダ抽象にのみ適用され、T-Var は変数にのみ適用されるなどのように、他のすべての型付け規則はある特定の形の項を指定しているが、T-Sub は**どんな**種類の項にも適用できてしまう。このことは、ある項 `t` が与えられてその型を計算したい場合に、規則 T-Sub と、残りの規則で `t` に合う結論を持つものと、両方が常に適用可能であることを意味する。

S-Trans に問題がある理由も同じで、この規則の結論は他のすべての規則の結論と重複する。

$$\frac{\mathtt{S} <: \mathtt{U} \qquad \mathtt{U} <: \mathtt{T}}{\mathtt{S} <: \mathtt{T}} \qquad \text{(S-Trans)}$$

`S` と `T` は裸のメタ変数なので、潜在的にはすべての部分型付け判断式の導出において、最後の規則として S-Trans を使うことができてしまう。つまり、部分型付け規則の単純な「下から上」の実装では、この規則（S-Trans）を試すべきか、今部分型関係にあるかどうか調べようとしている二つの型に適合するより特化された結論を持つ他の規則を試すべきかが、全くわからない。

S-Trans にはもう一つの問題がある。二つの前提ともメタ変数 `U` について触れているが、これは結論には現れない。この規則を下から上に単純に読むと、型 `U` を**推測し**、`S <: U` と `U <: T` を示そうとしなければならないことになる。推測できる `U` の数は無限個あるため、この戦略はほとんど成功する望みがない。

規則 S-Refl も他の部分型付け規則の結論と重複する。このことは T-Sub や S-Trans における問題に比べれば深刻ではない。この反射律の規則には前提がないため、もし示したい部分型付け判断式がこの規則に合うならば、直ちに示せるからである。それでも、これは型付け規則が構文主導でない原因の一つになっている。

これらの問題はすべて、普通の（または**宣言的な**）部分型関係と型付け関係を、**アルゴリズム的部分型**関係と**アルゴリズム的型付け**関係と呼ばれる、構文主導的な推論規則の集合からなる二つの新しい関係で置き換えれば解決できる。そして、元の部分型付け関係と型付け関係が実際にこれらのアルゴリズム的な表現と一致することを証明することで、この書き換えの正しさを示す。つまり、部分型付け判断式 `S <: T` がアルゴリズム的部分型付け規則から導出できるとき（かつそのときのみ）宣言的規則から導出できることと、ある項がアルゴリズム的型付け規則によって型付け可能であるとき（かつそのときのみ）宣言的規則の下で型付け可能であることを示す。

→ {} <: 図 15-1 と図 15-3 の拡張

$\mathsf{S <: S}$	(S-REFL)	$\dfrac{\mathsf{T_1 <: S_1} \quad \mathsf{S_2 <: T_2}}{\mathsf{S_1 \to S_2 <: T_1 \to T_2}}$	(S-ARROW)
$\dfrac{\mathsf{S <: U} \quad \mathsf{U <: T}}{\mathsf{S <: T}}$	(S-TRANS)	$\dfrac{\{\mathsf{l}_i{}^{i\in 1..n}\} \subseteq \{\mathsf{k}_j{}^{j\in 1..m}\} \quad \mathsf{k}_j = \mathsf{l}_i \text{ならば} \mathsf{S}_j <: \mathsf{T}_i}{\{\mathsf{k}_j\mathsf{:S}_j{}^{j\in 1..m}\} <: \{\mathsf{l}_i\mathsf{:T}_i{}^{i\in 1..n}\}}$	(S-RCD)
$\mathsf{S <: Top}$	(S-TOP)		

▲ 図 16-1. レコードのある部分型関係（簡潔な版）

16.1 節ではアルゴリズム的部分型関係を、16.2 節ではアルゴリズム的型付け関係を説明する。16.3 節では `if...then...else` のような複数に分岐する構文の特別な型検査問題に取り組む。これには新しい構造（部分型関係における上限または**結び**の存在）が必要になる。16.4 節では最小型 Bot について考える。

16.1 アルゴリズム的部分型付け

ある型が別の型の部分型かどうか検査するためのアルゴリズムは、部分型付けを持つ言語のどんな実装でも極めて重要な要素となる。この部分型検査器は、例えば $\mathsf{t_1}$ が $\mathsf{T \to U}$ 型を持ち、$\mathsf{t_2}$ が S 型を持つとして、関数適用 $\mathsf{t_1\ t_2}$ に出会ったとき、型検査器から呼び出される。この部分型検査器の機能は部分型付け判断式 $\mathsf{S <: T}$ が図 15-1 と図 15-3 の部分型付け規則から導出できるかどうかを決定することである。これを達成するため、$(\mathsf{S,T})$ が $\mapsto \mathsf{S <: T}$（「S はアルゴリズム的に T の部分型となる」）と書かれる、別の関係に属しているかどうか調べる。この関係は、型の構造に従えば簡単に所属が決定できるように定義され、かつ型の組の集合としては普通の部分型関係と一致する。宣言的な関係とアルゴリズム的な関係との重要な違いは、アルゴリズム的な関係には規則 S-TRANS と S-REFL がないということである。

まず、宣言的な型システムを少し再編成する必要がある。143 ページで見たように、深さ、幅、並べ替えの部分型付けの組み合わせを含んだレコードの部分型付け導出を「張り合わせる」ためには推移律を使う必要がある。S-TRANS をなくす前に、まず深さ、幅、並べ替えの部分型付けを一つにまとめる規則を追加しなければならない。

$$\frac{\{\mathsf{l}_i{}^{i\in 1..n}\} \subseteq \{\mathsf{k}_j{}^{j\in 1..m}\} \qquad \mathsf{k}_j = \mathsf{l}_i \text{ ならば } \mathsf{S}_j <: \mathsf{T}_i}{\{\mathsf{k}_j\mathsf{:S}_j{}^{j\in 1..m}\} <: \{\mathsf{l}_i\mathsf{:T}_i{}^{i\in 1..n}\}} \qquad \text{(S-RCD)}$$

補題 16.1.1. もし、S-RCDDEPTH と S-RCDWIDTH と S-RCDPERM の規則を含む（しかし S-RCD は含まない）部分型付け規則で $\mathsf{S <: T}$ が導出できるならば、これは規則 S-RCD を使っても（S-RCDDEPTH と S-RCDWIDTH と S-RCDPERM は使わないで）導出できる。また、逆も同様である。 □

[証明] 導出に関する単純な帰納法による。 □

補題 16.1.1 は規則 S-RCDDEPTH と S-RCDWIDTH と S-RCDPERM を取り除き、代わりに S-RCD を用いることが正当であることを示している。結果的に得られる型システムを図 16-1 に要約する。

次に、図 16-1 の型システムで反射律と推移律の規則が不要であることを示す。

$\rightarrow$ {} <:

アルゴリズム的部分型付け $\boxed{\mapsto S <: T}$

$$\mapsto S <: Top \quad \text{(SA-Top)}$$

$$\frac{\mapsto T_1 <: S_1 \qquad \mapsto S_2 <: T_2}{\mapsto S_1 \rightarrow S_2 <: T_1 \rightarrow T_2} \quad \text{(SA-Arrow)}$$

$$\frac{\{l_i\ ^{i\in 1..n}\} \subseteq \{k_j\ ^{j\in 1..m}\} \qquad k_j = l_i \text{ ならば } \mapsto S_j <: T_i}{\mapsto \{k_j : S_j\ ^{j\in 1..m}\} <: \{l_i : T_i\ ^{i\in 1..n}\}} \quad \text{(SA-Rcd)}$$

▲ 図 16-2. アルゴリズム的部分型付け

補題 16.1.2.

(1) すべての型 S について、S <: S は S-Refl を使わずに導出できる。
(2) もし S <: T が導出可能ならば、S <: T は S-Trans を使わずに導出できる。 □

[証明] 演習[推奨, ★★★]。 □

演習 16.1.3. [★] もし、型 Bool を追加したら、これらの性質はどう変化するだろうか。 □

このことからアルゴリズム的な部分型関係の定義がもたらされる。

定義 16.1.4. **アルゴリズム的部分型**関係は、図 16-2 の規則に閉じた最小の関係である。 □

アルゴリズム的規則で導出できるすべての部分型付け判断式が宣言的規則でも導出できる（アルゴリズム的規則はどんな新しいことも証明しない）という理由から、アルゴリズム的規則は**健全**であるといい、また、宣言的規則で導出できるすべての部分型付け判断式がアルゴリズム的規則でも導出できる（アルゴリズム的規則はこれまで証明できたすべてのことを証明できる）という理由から、アルゴリズム的規則は**完全**であるという。

命題 16.1.5. [**健全性と完全性**] S <: T のとき、かつそのときのみ $\mapsto$ S <: T □

[証明] どちらの方向も、導出に関する帰納法によって、前の二つの補題のうち一方を使って証明される。 □

これで、構文主導であるアルゴリズム的規則はアルゴリズム的部分型関係を検査する（すなわち宣言的部分型関係も検査する）アルゴリズムとして読める。より慣習的な擬似コードの記法では、このアルゴリズムは次のようになる。

$$
\begin{array}{lll}
\textit{subtype}(S,T) & = & \text{if } T = Top \text{ then } \textit{true} \\
 & & \text{else if } S = S_1 \rightarrow S_2 \text{ and } T = T_1 \rightarrow T_2 \\
 & & \quad \text{then } \textit{subtype}(T_1, S_1) \wedge \textit{subtype}(S_2, T_2) \\
 & & \text{else if } S = \{k_j : S_j\ ^{j\in 1..m}\} \text{ and } T = \{l_i : T_i\ ^{i\in 1..n}\} \\
 & & \quad \text{then } \{l_i\ ^{i\in 1..n}\} \subseteq \{k_j\ ^{j\in 1..m}\} \\
 & & \qquad \wedge \text{ すべての } i \text{ に対して、} j \in 1..m \text{ が存在し、} k_j = l_i \\
 & & \qquad\quad \text{かつ } \textit{subtype}(S_j, T_i) \\
 & & \text{else } \textit{false}
\end{array}
$$

このアルゴリズムの ML による具体的な実現は第 17 章に現れる。

最後にアルゴリズム的部分型関係が**全域的**、つまり、アルゴリズム的規則から導出されたこの再帰関数 *subtype* が、すべての入力の二つ組について、有限時間内に *true* または *false* を返すということを検証する必要がある。

命題 16.1.6.［停止性］ もし ↦ S <: T が導出可能ならば、*subtype*(S, T) は *true* を返す。そうでなければ、*subtype*(S, T) は *false* を返す。 □

この定理は、アルゴリズム的規則の健全性および完全性と併せると、関数 *subtype* が宣言的部分型関係のための**決定手続き**であることを本質的に主張している。

［証明］ 最初の主張は簡単に確かめられる（↦ S <: T の導出に関する単純な帰納法による）。逆に、*subtype*(S, T) が *true* を返すならば、↦ S <: T となることを示すことも簡単である。したがって、二つ目の主張を証明するためには、*subtype*(S, T) が常に**何か**を返す、つまり発散しえないことを示せば十分である。このことは、入力の二つ組 S と T の大きさの和が、アルゴリズムにおけるどの再帰呼び出しの実引数の大きさの和よりも真に大きくなることを確認することで示せる。この和は常に正なので再帰呼び出しの無限列は存在しえない。 □

読者は、部分型関係のアルゴリズム的定義を正式なものとして扱い、宣言的定義には全く言及さえしないことで、この節でのすべての仕事を省けないだろうかと思うかもしれない。答えは、条件付きで「その通り」である。望むならば、計算体系を定義する際に、アルゴリズム的定義を正式のものとして扱うことは確かにできる。しかし、それでも実際はそれほど多くの仕事が減るわけではない。なぜなら、型付け関係（これは部分型関係に依存している）がよく振る舞うことを示すために、部分型付けが反射的かつ推移的であることを知る必要があり、それらの証明は、これまで行ったこととだいたい同じくらいの仕事になるからである。（他方では、言語の定義はしばしば型付け関係のアルゴリズム的な表現を採用することが確かにある。第19章でこの一例を扱う。）

16.2 アルゴリズム的型付け

部分型関係をアルゴリズム的にした今、型付け関係もアルゴリズム的にする必要がある。163ページで見たように、T-SUB だけが構文主導でない型付け規則であり、まさにこの規則に取り組まなければならない。前節における S-TRANS と同じく、単純に包摂規則を消すことはできない。すなわち、型付けにおいて、どこでそれが重要な役割を果たすかどうかを最初に調べ、より構文主導的な方法で同じ効果が得られるよう、残りの型付け規則を強化しなければならない。

明らかに、包摂の重要な用途の一つは、関数によって期待される型と、引数の実際の型との間の隔たりを埋めることである。次のような項は包摂なしには型付けすることができない。

```
(λr:{x:Nat}. r.x) {x=0,y=1}
```

驚くことかもしれないが、これは包摂が型付けにおいて重大な役割を果たす**唯一の**状況である。型付けの証明で包摂を使っている他のすべての場合では、同じ部分型付け判断式を別の導出によって証明できる。包摂を木の根に向かって動かし「後回しにする」導出である。なぜこれがうまくいくのかを見るために、次のようなちょっとした試みが有益である。包摂を含む型付け導出について、それぞれの型付け規則を順番に選び、もしその直接の部分導出の一つが T-SUB で終わっているならば、この規則で終わる導出はどのように再編成できるかについて考えてみよう。

例えば、T-ABS で終わる型付け導出で、直接の部分導出が T-SUB で終わっているようなものが与えられたとする。

$$\dfrac{\dfrac{\dfrac{\vdots}{\Gamma, x{:}S_1 \vdash s_2 : S_2} \qquad \dfrac{\vdots}{S_2 <: T_2}}{\Gamma, x{:}S_1 \vdash s_2 : T_2}\,\text{(T-Sub)}}{\Gamma \vdash \lambda x{:}S_1.s_2 : S_1\to T_2}\,\text{(T-Abs)}$$

そのような導出は、ラムダ抽象の規則の**後に**包摂を使って同じ結論に達するように再配置できる。

$$\dfrac{\dfrac{\dfrac{\vdots}{\Gamma, x{:}S_1 \vdash s_2 : S_2}}{\Gamma \vdash \lambda x{:}S_1.s_2 : S_1\to S_2}\,\text{(T-Abs)} \qquad \dfrac{\dfrac{}{S_1 <: S_1}\,\text{(S-Refl)} \quad \dfrac{\vdots}{S_2 <: T_2}}{S_1\to S_2 <: S_1\to T_2}\,\text{(S-Arrow)}}{\Gamma \vdash \lambda x{:}S_1.s_2 : S_1\to T_2}\,\text{(T-Sub)}$$

より面白いのは関数適用の規則 T-App の場合である。ここでは、二つの部分導出が存在し、どちらも T-Sub で終わるかもしれない。まずは左側の部分導出の最後に包摂が現れるような場合を考えよう。

$$\dfrac{\dfrac{\dfrac{\vdots}{\Gamma \vdash s_1 : S_{11}\to S_{12}} \qquad \dfrac{\vdots}{S_{11}\to S_{12} <: T_{11}\to T_{12}}\,\text{(S-Arrow)}}{\Gamma \vdash s_1 : T_{11}\to T_{12}}\,\text{(T-Sub)} \qquad \dfrac{\vdots}{\Gamma \vdash s_2 : T_{11}}}{\Gamma \vdash s_1\ s_2 : T_{12}}\,\text{(T-App)}$$

前の章での結果から、$S_{11}\to S_{12} <: T_{11}\to T_{12}$ の導出における最後の規則が S-Refl でも S-Trans でもないと仮定してもよい。結論の形から、この規則は S-Arrow でしかありえない。

$$\dfrac{\dfrac{\dfrac{\vdots}{\Gamma \vdash s_1 : S_{11}\to S_{12}} \qquad \dfrac{\dfrac{\vdots}{T_{11} <: S_{11}} \quad \dfrac{\vdots}{S_{12} <: T_{12}}}{S_{11}\to S_{12} <: T_{11}\to T_{12}}\,\text{(S-Arrow)}}{\Gamma \vdash s_1 : T_{11}\to T_{12}}\,\text{(T-Sub)} \qquad \dfrac{\vdots}{\Gamma \vdash s_2 : T_{11}}}{\Gamma \vdash s_1\ s_2 : T_{12}}\,\text{(T-App)}$$

T-Sub のインスタンスを消す書き換えには面白い効果がある。

$$\dfrac{\dfrac{\dfrac{\vdots}{\Gamma \vdash s_1 : S_{11}\to S_{12}} \qquad \dfrac{\dfrac{\vdots}{\Gamma \vdash s_2 : T_{11}} \quad \dfrac{\vdots}{T_{11} <: S_{11}}}{\Gamma \vdash s_2 : S_{11}}\,\text{(T-Sub)}}{\Gamma \vdash s_1\ s_2 : S_{12}}\,\text{(T-App)} \qquad \dfrac{\vdots}{S_{12} <: T_{12}}}{\Gamma \vdash s_1\ s_2 : T_{12}}\,\text{(T-Sub)}$$

元の S-Arrow のインスタンスの右側の部分導出は、木の一番下に押し下げられ、そこで新しい T-Sub のインスタンスが関数適用ノード全体の型を昇格している。一方で、元の T-App のインスタンスの右側の部分導出は引数 s_2 の導出の中へ**押し上げ**られた。

変わって今度は、移動させたい T-Sub のインスタンスが T-App のインスタンスの右側の部分導出の最後に現れると仮定する。

$$\dfrac{\dfrac{\vdots}{\Gamma \vdash s_1 : T_{11}\to T_{12}} \qquad \dfrac{\dfrac{\vdots}{\Gamma \vdash s_2 : T_2} \quad \dfrac{\vdots}{T_2 <: T_{11}}}{\Gamma \vdash s_2 : T_{11}}\,\text{(T-Sub)}}{\Gamma \vdash s_1\ s_2 : T_{12}}\,\text{(T-App)}$$

T-Sub のこのインスタンスに対してできるのは左側の部分導出へと動かすことだけである。つまり、上の変換を部分的に反転させるのである。

$$\dfrac{\dfrac{\dfrac{\vdots}{\Gamma \vdash s_1 : T_{11}\to T_{12}} \qquad \dfrac{\dfrac{\vdots}{T_2 <: T_{11}} \quad \dfrac{}{T_{12} <: T_{12}}\,\text{(S-Refl)}}{T_{11}\to T_{12} <: T_2\to T_{12}}\,\text{(S-Arrow)}}{\Gamma \vdash s_1 : T_2\to T_{12}}\,\text{(T-Sub)} \qquad \dfrac{\vdots}{\Gamma \vdash s_2 : T_2}}{\Gamma \vdash s_1\ s_2 : T_{12}}\,\text{(T-App)}$$

関数適用の結果の型を昇格するための包摂の使用は規則 T-App よりも下に動かせることを見た。しかし、引数の型と関数の定義域の型を適合させるための包摂の使用は消すことができない。それは、一方の前提から他方へ動かすことはできる。つまり、引数の型を関数の定義域に適合するように昇格するか、または関数の型を（その引数の型を降格することによって）昇格し実際に与えようとしている型の引数を受け取るようにできる。しかし、完全にこの包摂をなくすことはできない。この観察は、このようなギャップを埋める形での包摂の使用が、この型システムの能力の本質であるという直観に精密に対応する。

もう一つ考えなければならないのは、導出の最後の規則が包摂で、かつ、その直接の部分導出も包摂で終わっている場合である。この場合、二つの隣り合った包摂の使用は一つに合体できる。すなわち、次の形をした任意の導出は、

$$\dfrac{\dfrac{\dfrac{\vdots}{\Gamma \vdash s : S} \quad \dfrac{\vdots}{S <: U}}{\Gamma \vdash s : U}\,\text{(T-Sub)} \qquad \dfrac{\vdots}{U <: T}}{\Gamma \vdash s : T}\,\text{(T-Sub)}$$

次のように書き換えられる。

$$\dfrac{\dfrac{\vdots}{\Gamma \vdash s : S} \qquad \dfrac{\dfrac{\vdots}{S <: U} \quad \dfrac{\vdots}{U <: T}}{S <: T}\,\text{(S-Trans)}}{\Gamma \vdash s : T}\,\text{(T-Sub)}$$

演習 16.2.1. [★ ↛]　この実験を仕上げるため、T-Rcd または T-Proj の前で規則 T-Sub が使われているような導出において、どのように同様の再配置を実行すればよいかを示せ。 □

これらの変換を繰り返し行うことで、任意の型付け導出を T-Sub が二つの場所だけに現れる特別な形に書き換えられる。それは関数適用の右側の部分導出の最後と、導出全体の一番最後である。それに加えて、もし単純に一番最後のほうを**消した**としても、重大な害は起こらない。つまり、まだ同じ項に型を割り当てる導出があるはずである。唯一の違いは、この項に割り当てられた型がより小さい（つまり、もっと良い）型になるかもしれないことである。このことから、包摂が使われうる場所はただ一つ、関数適用だけになる。この場合に対応するために、関数適用の規則を少し強力なもので置き換えればよい。

$\rightarrow$ {} <:

アルゴリズム的型付け $\boxed{\Gamma \mapsto t : T}$

$$\frac{x{:}T \in \Gamma}{\Gamma \mapsto x : T} \quad \text{(TA-VAR)}$$

$$\frac{\Gamma, x{:}T_1 \mapsto t_2 : T_2}{\Gamma \mapsto \lambda x{:}T_1.t_2 : T_1 \to T_2} \quad \text{(TA-ABS)}$$

$$\frac{\Gamma \mapsto t_1 : T_1 \qquad T_1 = T_{11} \to T_{12} \qquad \Gamma \mapsto t_2 : T_2 \qquad \mapsto T_2 <: T_{11}}{\Gamma \mapsto t_1\ t_2 : T_{12}} \quad \text{(TA-APP)}$$

$$\frac{\text{各 } i \text{ に対して } \Gamma \mapsto t_i : T_i}{\Gamma \mapsto \{l_1{=}t_1 \ldots l_n{=}t_n\} : \{l_1{:}T_1 \ldots l_n{:}T_n\}} \quad \text{(TA-RCD)}$$

$$\frac{\Gamma \mapsto t_1 : R_1 \qquad R_1 = \{l_1{:}T_1 \ldots l_n{:}T_n\}}{\Gamma \mapsto t_1.l_i : T_i} \quad \text{(TA-PROJ)}$$

▲ 図 16-3. アルゴリズム的型付け

$$\frac{\Gamma \vdash t_1 : T_{11} \to T_{12} \qquad \Gamma \vdash t_2 : T_2 \qquad T_2 <: T_{11}}{\Gamma \vdash t_1\ t_2 : T_{12}}$$

前提に包摂のインスタンスを一つ合体させるのである。包摂に続く関数適用の形をしたすべての部分導出は、この規則の使用に置き換えられ、規則 T-SUB を全く使わない形になる。さらに、強力になった関数適用規則は構文主導である。つまり、規則の結論に現れる項の形が、残りの規則と重複することを防いでいる。

この変換により、元の型システムと同じ項の集まりに型を割り当てる構文主導的な型付け規則の集合が得られる。これらの規則は次の定義で要約される。アルゴリズム的部分型付け規則で行ったように、宣言的関係と区別するために、アルゴリズム的関係は、風変わりなターンスタイルを使って、$\Gamma \mapsto t : T$ のように記述する。

定義 16.2.2. アルゴリズム的型付け関係は図 16-3 の規則について閉じた最小の関係である。関数適用の規則の前提 $T_1 = T_{11} \to T_{12}$ は、単に型検査中の処理の順番を明示しているだけである。つまり、まず t_1 の型 T_1 を計算し、そして T_1 が $T_{11} \to T_{12}$ の形をしているか確かめる、といった具合である。この規則は、その前提をなくし、代わりに最初の前提を $\Gamma \mapsto t_1 : T_{11} \to T_{12}$ のように書き換えたものと全く同じ力を持つだろう。TA-PROJ についても同様である。関数適用規則に現れる部分型付けの前提も、風変わりなターンスタイルを使って書かれている。部分型付け規則において、アルゴリズム的なほうと宣言的なほうが等価であることがわかっているので、この選択は好みの問題である。 □

演習 16.2.3. [★ ↛] ある項にアルゴリズム的型付け規則で割り当てられる型が評価によって**減少**しうることを示せ。これは、ある二つの項 s と t で、アルゴリズム的型付けで S と T に型付けされ、$s \longrightarrow^* t$ かつ $T <: S$ だが、$S \not<: T$ となるようなものを見つけることで示せ。 □

ここで、アルゴリズム的型付け規則が元の宣言的規則に相当することを形式的に確認することが必要である。(上で列挙した型付け導出の変換は、証明とみなすには非形式的すぎる。これらは証明の形にできるかもしれないが、必要以上に長くて重くなってしまうだろう。いつものように、単に導出に関する帰納法を使って議論するのが、より単純である。) 部分型付けのときと同様に、アルゴリズム的型付け関係が、元の宣言的な規則について、健全かつ完全であることを議論する。

健全性の性質は変わらない。アルゴリズム的規則から導出できるすべての型付け判断式は、宣言的規則からも導出可能である。

定理 16.2.4.［健全性］　$\Gamma \mapsto$ t : T ならば $\Gamma \vdash$ t : T となる。□

［証明］　アルゴリズム的型付け導出に関する単純な帰納法により示せる。□

完全性は少し異なる見た目になる。普通の型付け関係は一つの項に多数の型を割り当てるために使えるが、一方アルゴリズム的型付け関係は高々一つの型を割り当てる（このことは容易に確認できる）。よって、定理 16.2.4 を単に逆にしても、明らかに成り立たない。その代わりに、普通の型付け規則の下で、項 t が型 T を持つならば、その項は S <: T という意味で**より良い**型 S をアルゴリズム的規則の下で持つ、ということを示せる。つまり、アルゴリズム的規則は型付け可能な各々の項に可能な限り小さい型（**最小**型）を割り当てるのである。完全性定理はしばしば最小型付け定理と呼ばれる。それは、（定理 16.2.4 と組み合わせたとき）宣言的な型システムで型付け可能な各項が、最小型を持つことを証明することになるからである。

定理 16.2.5.［完全性、最小型付け］　もし $\Gamma \vdash$ t : T ならば、ある S <: T が存在して、$\Gamma \mapsto$ t : S となる。□

［証明］　演習［推奨, ★★］。□

演習 16.2.6.［★★］　もし部分型付け規則 S-Arrow をなくし、それ以外の宣言的な部分型付け規則および型付け規則はそのままにした場合、この体系は最小型付けの性質を持つか。もしそうならば、それを証明せよ。そうでないならば最小型を持たない、型付け可能な項の例を示せ。□

16.3　結びと交わり

部分型付けのある言語において、条件式や case 式のような幾つかの結果の分岐を持つ式を型検査するには、いくらか追加の機構が必要である。例えば、条件式の宣言的な型付け規則を思い出そう。

$$\frac{\Gamma \vdash t_1 : \mathtt{Bool} \qquad \Gamma \vdash t_2 : \mathtt{T} \qquad \Gamma \vdash t_3 : \mathtt{T}}{\Gamma \vdash \mathtt{if}\ t_1\ \mathtt{then}\ t_2\ \mathtt{else}\ t_3 : \mathtt{T}} \qquad \text{(T-If)}$$

この規則は、二つの分岐の型が同じであることを要求し、そしてその型を条件式全体の型に割り当てる。しかし、包摂がある場合、二つの分岐に同じ型を付ける**多くの**方法があるかもしれない。例えば、

```
if true then {x=true,y=false} else {x=false,z=true};
```

は {x:Bool} という型を持つ。なぜなら、then 節は最小型 {x:Bool,y:Bool} を持ち、これは T-Sub によって {x:Bool} に昇格でき、同様に else 節も {x:Bool, z:Bool} を持ち、{x:Bool} 型に昇格できるからである。この項は {x:Top} や {} 型も持つ。実際これは、{x:Bool,y:Bool} と {x:Bool,z:Bool} の両方の上位型である**任意の**型を持つ。よって、条件式全体の最小型は {x:Bool,y:Bool} と {x:Bool,z:Bool} の両方の上位型となる最小の型、つまり {x:Bool} である。一般的に、任意の条件式の最小型を計算するためには、then 節と else 節の最小型をそれぞれ計算し、これらに共通する上位型のうち最小のものを計算する必要がある。その型は分岐の型につ

いての**結び**としばしば呼ばれる。これは半順序における二つの要素の結びに相当するからである。

定義 16.3.1. 型の二つ組 S、T に対して、ある型 J が、S <: J かつ T <: J かつ、任意の型 U に対して、S <: U かつ T <: U ならば J <: U となるとき、J を S と T の**結び**と呼び、S ∨ T = J と書く。同様に、ある型 M が、M <: S かつ M <: T かつ、任意の型 L に対して、L <: S かつ L <: T ならば、L <: M となるとき、M を S と T の**交わり**と呼び、S ∧ T = M と書く。 □

部分型関係がその部分型付けを持つ言語においてどのように定義されているかに依存して、すべての型の二つ組が結びを持つこともあり、そうでないこともある。ある部分型関係について、すべての S と T に対して S と T の結びとなる J が存在するとき、この部分型関係は**結びを持つ**という。同様に、すべての S と T に対して S と T の交わり M が存在するとき、この部分型関係は**交わりを持つ**という。

この節で考える部分型関係[†1]は結びを持つが交わりを持たない。例えば型 {} と Top→Top は共通の部分型を全く持たないので、確実に最大の共通の部分型は存在しない。しかし、少しだけ弱めた性質は満たす。型の二つ組 S と T が、ある型 L が存在して L <: S かつ L <: T となるとき、**下に有界**であるという。下に有界であるすべての S と T に対して、S と T の交わり M が存在するような部分型関係は**有界な交わりを持つ**という。

結びや交わりは唯一である必要はない。例えば、{x:Top,y:Top} と {y:Top,x:Top} は両方とも、型の二つ組 {x:Top,y:Top,z:Top} と {x:Top,y:Top,w:Top} の結びである。しかし、同じ型の二つ組に対する、二つの異なる結び（または交わり）は必ず、お互いの部分型となる。

命題 16.3.2. [**結びと有界な交わりの存在**]

(1) すべての型の二つ組 S と T について、ある型 J が存在して、S ∨ T = J となる。
(2) 共通の部分型を持つ、すべての型の二つ組 S と T について、ある型 M が存在して、S ∧ T = M となる。 □

[証明] 演習[推奨, ★★★]。 □

結び演算を使って、部分型付けの下での if 構文のためのアルゴリズム的規則を与えることができる。

$$\frac{\Gamma \vdash \mathtt{t}_1 : \mathtt{T}_1 \qquad \mathtt{T}_1 = \mathtt{Bool} \qquad \Gamma \vdash \mathtt{t}_2 : \mathtt{T}_2 \qquad \Gamma \vdash \mathtt{t}_3 : \mathtt{T}_3 \qquad \mathtt{T}_2 \vee \mathtt{T}_3 = \mathtt{T}}{\Gamma \vdash \mathtt{if}\ \mathtt{t}_1\ \mathtt{then}\ \mathtt{t}_2\ \mathtt{else}\ \mathtt{t}_3 : \mathtt{T}} \quad \text{(TA-IF)}$$

演習 16.3.3. [★★] if true then false else {} の最小型は何か。これは我々の求めているものか。 □

演習 16.3.4. [★★★] 結びと交わりを計算するアルゴリズムを、15.5 節（151 ページ）で記述されたような参照を持つ手続き型言語へ拡張することは簡単か。非変な Ref を共変な Source と反変な Sink に詳細化した 15.5 節（154 ページ）での参照の扱いについてはどうか。 □

---

†1 つまり、図 15-1 と図 15-3 で定義した関係を、Bool 型で拡張をしたものである。Bool についての部分型付けは単純である。宣言的部分型関係には Bool についての規則は追加しないので、その上位型は Top だけとなる。

16.4 アルゴリズム的型付けと Bottom 型

もし部分型関係に最小型 Bot（15.4 節）が加えられたならば、部分型付けと型付けのアルゴリズムは少し拡張をしなければならない。アルゴリズム的部分型関係に一つの規則（明らかな規則）を追加する。

$$\mapsto \mathsf{Bot} <: \mathsf{T} \qquad \text{(SA-BOT)}$$

そして、アルゴリズム的型付け関係には、もう少しだけ巧妙な二つの規則を追加する。

$$\frac{\Gamma \mapsto \mathsf{t}_1 : \mathsf{T}_1 \qquad \mathsf{T}_1 = \mathsf{Bot} \qquad \Gamma \mapsto \mathsf{t}_2 : \mathsf{T}_2}{\Gamma \mapsto \mathsf{t}_1\ \mathsf{t}_2 : \mathsf{Bot}} \qquad \text{(TA-APPBOT)}$$

$$\frac{\Gamma \mapsto \mathsf{t}_1 : \mathsf{R}_1 \qquad \mathsf{R}_1 = \mathsf{Bot}}{\Gamma \mapsto \mathsf{t}_1.\mathsf{l}_i : \mathsf{Bot}} \qquad \text{(TA-PROJBOT)}$$

部分型付け規則は明らかである。型付け規則の背景にある直観は、宣言的な型システムでは、Bot 型のものには無条件に任意の型の引数を適用できる、ということである（これは包摂を使って、Bot をどんな関数型にでもできるからである）。そして、結果は任意の他の型を持つことを仮定できる。射影に関しても同様である。

演習 16.4.1. [★] この言語が条件式も持っていたとする。このとき、if についてのアルゴリズム的型付け規則を追加する必要があるだろうか。 □

この言語で Bot をサポートするために必要な追加分は、そこまで複雑ではない。しかし、28.8 節で、Bot が有界量化と合わさったとき、より深刻な複雑さが発生することがわかるだろう。

部分型付けの ML 実装

この章では、第 10 章で作成した単純型付きラムダ計算の OCaml での実装を拡張し、部分型付けの実現に必要な付加的機構、特に、部分型関係を検査する関数を付け加える。

17.1 構文

型と項のデータ型定義は図 15-1 および図 15-3 に示される抽象構文に従う。

```
type ty =
    TyRecord of (string * ty) list
  | TyTop
  | TyArr of ty * ty

type term =
    TmVar of info * int * int
  | TmAbs of info * string * ty * term
  | TmApp of info * term * term
  | TmRecord of info * (string * term) list
  | TmProj of info * term * string
```

純粋な単純型付きラムダ計算と比べて新規の構築子は、型 TyTop、型構築子 TyRecord、および項の構築子 TmRecord と TmProj である。レコードとその型は可能な限り単純な方法で、すなわち、フィールド名とそれに関連する項または型のリストとして表すことにする。

17.2 部分型付け

165 ページにあるアルゴリズム的部分型関係の擬似コードは、以下のように、OCaml に直接翻訳できる。

```
let rec subtype tyS tyT =
   (=) tyS tyT ||
   match (tyS,tyT) with
     (_,TyTop) →
       true
   | (TyArr(tyS1,tyS2),TyArr(tyT1,tyT2)) →
       (subtype tyT1 tyS1) && (subtype tyS2 tyT2)
   | (TyRecord(fS), TyRecord(fT)) →
       List.for_all
         (fun (li,tyTi) →
            try let tySi = List.assoc li fS in
                subtype tySi tyTi
            with Not_found → false)
         fT
   | (_,_) →
       false
```

擬似コードから一か所わずかな変更をアルゴリズムに施し、冒頭に反射性の検査を加えている。((=) 演算子は通常の等号である。ここで前置で書くようにしているのは、他の部分型付けの実装では異なる比較用関数の呼び出しに置き換えられる場合があるからである。|| 演算子は短絡評価による論理和[†1]である。第一引数が true ならば、第二引数が評価されることはない。) 厳密に言えば、この検査は必要ではない。しかし実際には、現実のコンパイラにとっては重要な最適化である。現実世界でのプログラムの大多数では、部分型付けが利用されることは極めて稀である。つまり、部分型検査器が呼ばれるとき、比較対象の二つの型は実際には等しいことがほとんどなのである。さらに、もし、構造的に同型な型が物理的に同一の表現を持つように型が表現される場合（例えば、型の構築に**ハッシュコンシング**（Goto [184]; Appel and Gonçalves [20]）を用いる場合)、この検査は 1 命令で済む。

レコードの部分型付け規則は、自然とある程度、リスト操作の手間がかかる。List.for_all は述語（第一引数）をリストのすべての要素に適用し、すべての適用結果が true だったとき、かつそのときのみ true を返す。List.assoc li fS は、フィールド fS のリスト内にあるラベル li を検索し、関連付けられたフィールドの型 tySi を返す。もし li が fS 内のラベルでないならば Not_found を投げる。この例外はキャッチされて返り値 false に変換される。

17.3 型付け

型検査関数は、先の実装での typeof 関数を単純に拡張したものである。主な変更は関数適用の節で、実引数の型と、関数が期待する型の間で部分型の検査を行う。また、レコードの構築と射影のために、新たに二つの節を加える。

```
let rec typeof ctx t =
  match t with
    TmRecord(fi, fields) →
      let fieldtys =
        List.map (fun (li,ti) → (li, typeof ctx ti)) fields in
      TyRecord(fieldtys)
  | TmProj(fi, t1, l) →
      (match (typeof ctx t1) with
          TyRecord(fieldtys) →
            (try List.assoc l fieldtys
             with Not_found → error fi ("label "^l^" not found"))
        | _ → error fi "Expected record type")
  | TmVar(fi,i,_) → getTypeFromContext fi ctx i
  | TmAbs(fi,x,tyT1,t2) →
      let ctx' = addbinding ctx x (VarBind(tyT1)) in
      let tyT2 = typeof ctx' t2 in
      TyArr(tyT1, tyT2)
  | TmApp(fi,t1,t2) →
      let tyT1 = typeof ctx t1 in
      let tyT2 = typeof ctx t2 in
      (match tyT1 with
          TyArr(tyT11,tyT12) →
            if subtype tyT2 tyT11 then tyT12
            else error fi "parameter type mismatch"
        | _ → error fi "arrow type expected")
```

†1 （訳注）第一引数の値が真ならば、第二引数を評価せず、真を返す論理和演算のこと。

レコードの節は、これまで見なかったOCamlの機能を幾つか導入している。TmRecord 節では、フィールド名と型のリスト fieldtys をフィールド名と項のリスト fields から求める。これは List.map を用いて、関数

```
fun (li,ti) → (li, typeof ctx ti)
```

を各名前・項の組に順に適用することで計算する。TmProj 節では、List.assoc を再び使って選択したフィールドの型を検索する。Not_found が発生したら、独自のエラーメッセージを出力する(^ は文字列の連結である)。

演習 17.3.1. [★★★] 16.3 節では、部分型付けのある言語にブール値と条件式を加える際、なぜ与えられた二つの型の上限を求める追加の補助関数が必要になるかを示した。命題 16.3.2 の証明(413 ページを見よ)では、必要となるアルゴリズムの数学的記述を与えた。

型検査器 joinexercise は、部分型、レコード、および条件式を備えた、単純型付きラムダ計算の不完全な実装である。基本的な構文解析と表示の関数は備わっているが、TmIf 用の節が typeof 関数から消え、それが依存する join 関数も同様になくなっている。ブール値と条件式(および結びと交わり)をこの実装に加えよ。 □

演習 17.3.2. [★★] 16.4 節の記述に従って、最小の型 Bot を実装 rcdsub に加えよ。 □

演習 17.3.3. [★★★ ↛] 関数適用の型付け規則において部分型検査が失敗したときに、我々の型検査器が表示するエラーメッセージは、ユーザにとってあまり役に立たないかもしれない。期待される引数の型と実引数の型をエラーメッセージ中に含めるように改善することもできるが、そのようなエラーメッセージでも理解するのは難しいかもしれない。例えば、期待される型が

```
{x:{},y:{},z:{},a:{},b:{},c:{},d:{},e:{},f:{},g:{}}
```

で、実際の型が

```
{y:{},z:{},f:{},a:{},x:{},i:{},b:{},e:{},g:{},c:{},h:{}}
```

だった場合、二つ目の型から欠けているのがフィールド d であるとは、すぐにはわからない。エラーの報告は、subtype 関数に変更を加え、true を返す代わりに自明な値(unit 値 ())を返し、false を返す代わりに例外を投げるようにすれば大幅に改善されうる。例外は、何かしらが実際にマッチしなかった型の箇所で発生するので、問題が何であったかについてより正確になりうる。この変更は検査器の振る舞いに「端から端まで」影響を与えるものではないことに注意されたい。部分型検査器が偽を返したら、どのみち型検査器は(error を呼んで)その箇所で常に例外を投げるからである。

関数 typeof と subtype を再実装し、**すべての**エラーメッセージが可能な限り有益な情報を持つようにせよ。 □

演習 17.3.4. [★★★ ↛] 15.6 節では、レコードと部分型付けを持つ言語に対し、型付けと部分型付け導出を純粋な単純型付きラムダ計算へ変換することによる**型強制意味論**を定義した。これらの変換を実装せよ。ここで、上述の subtype 関数を修正して(term として表現された)型強制関数を構築して返すようにし、同様に typeof 関数を修正して型と変換された項の両方を返すようにせよ。(元の入力された項でなく)変換後の項が評価されるようにし、結果を通常通り表示せよ。

□

第18章

事例：命令的オブジェクト

本章の例は、部分型付け（図15-1）、レコード（図15-3）、参照（図13-1）を伴った単純型付きラムダ計算の項である。関連するOCaml実装は`fullref`である。

本章ではじめて、実質のあるプログラミングの例を扱う。今まで定義してきた機能のほとんど（関数、レコード、一般的再帰、書き換え可能な参照、部分型付け）を用いて、SmalltalkやJavaといったオブジェクト指向言語に見られるようなオブジェクトやクラスを実現する、一連のプログラミングイディオムを構築する。本章では、それらのための新たな具象構文は全く導入しない。低水準の構文を用いて振る舞いを近似する方法を示すことで、オブジェクトやクラスといった、かなり複雑な言語機能を**理解**することを目的とする。

本章の大部分において、その近似は実際とても正確に行われる。既に見てきた機能の単純な組み合わせに脱糖衣される派生形式とみなすことで、オブジェクトやクラスのほとんどの機能の満足な実装が得られる。しかし、18.9節において仮想メソッドや`self`について触れるときは、評価順序に関する問題があり、脱糖衣が少し非現実的になってしまう。これらの機能を満足に説明するためには、第19章のように構文、操作的意味論、型付け規則を直接公理化すればよい。

18.1 オブジェクト指向プログラミングとは何か？

「〜の本質は何か？」についての論争はほとんどの場合、討論のトピックについて客観的事実を明らかにするよりも、討論の参加者の偏見を明らかにする。「オブジェクト指向」という用語を正確に定義しようとする試みも例外ではない。しかし、ほとんどのオブジェクト指向言語に見られる基礎的な機能で、同時に、長所も短所もよく知られた特徴的なプログラミングスタイルを可能にする機能を幾つか特定することはできる。

(1) **複数の表現**。オブジェクト指向の最も基本的な特徴はおそらく、操作がオブジェクトに対して呼び出されたとき、オブジェクト自身が、どのコードを実行するかを決めることにある。応答できる操作が同じ（つまり同じ**インターフェイス**を持つ）二つのオブジェクトは全く異なる表現を持てる。ただし各オブジェクトは、個々の表現で正しく動作するような、その操作の実装を持たなくてはならない。このような実装は、オブジェクトの**メソッド**と呼ばれる。オブジェクトに対する操作の呼び出し（**メソッド呼び出し**という。派手な言い方をすればオブジェクトへの**メッセージ**の送信とも）は実行時、オブジェクトに関連付けられたメソッドテーブルから操作名を参照する。この処理は**動的ディスパッチ**と呼ばれる。
対照的に、伝統的な**抽象データ型**（ADT）は、値の集合に、その値に対する操作の**単一の**実装を付け加えたものからなる。（このように実装を静的に定義することはオブジェクトと比べて利点も欠点もある。293ページでさらに探究する。）

(2) **カプセル化**。オブジェクトの内部表現は一般に、オブジェクトの定義の外からは見えないよう隠蔽される。つまり、オブジェクト自身のメソッドだけが、オブジェクトのフィールドを直

接参照したり操作したりできる[†1]。このことが意味するのは、オブジェクトの内部表現への変更は、プログラム中でも小さく容易に特定できる領域だけにしか影響しないということである。この制約は、大規模システムの可読性と保守性とを大きく改善する。

抽象データ型も同様のある種のカプセル化を提供し、値の具象表現が特定のスコープ（例えば、モジュールやADT定義）の中だけで見えること、そしてスコープ外のコードが値を操作するには、この特権的なスコープ内で定義された操作を呼び出すしかないことを保証する。

(3) **部分型付け**。オブジェクトの型（つまり、**インターフェイス**）は、オブジェクトの操作の名前と型の集まりにすぎない。オブジェクトの内部表現は型に現れ**ない**。オブジェクトに対して直接できる事柄に内部表現は影響を及ぼさないからである。

オブジェクトのインターフェイスは部分型関係に自然に適合する。もしオブジェクトがインターフェイス `I` を満たすならば、`I` よりも少ない操作を持つ任意のインターフェイス `J` を満たすのは明らかである。なぜなら、`J` のオブジェクトを期待する任意の文脈は、`J` の操作だけを呼び出せるので、`I` のオブジェクトを渡すのは常に安全だからである。（したがって、オブジェクトの部分型付けはレコードの部分型付けに似ている。実際、本章で説明するオブジェクトのモデルでは、その二つは同一のものとなる。）オブジェクトのインターフェイスの一部を無視することで、（ある一群の共通操作しか要求されない場合には）単一のコード片で多数の異なる種類のオブジェクトを統一的に扱う、といったことが可能になる。

(4) **継承**。インターフェイスを共有するオブジェクトはしばしば、幾つかの振る舞いも共有する。それゆえ、共通した振る舞いを実装するのは一度だけにしたい。ほとんどのオブジェクト指向言語は、このような振る舞いの再利用を、**クラス**と呼ばれる構造、および、**サブクラス化**を通して実現している。ここでクラスとは、オブジェクトがインスタンス化される元となるひな形のことである。また、サブクラス化とは、新しいメソッドの実装を追加したり、必要に応じて選択的に古いメソッドをオーバーライドしたりすることによって、古いクラスから新しいクラスを得る仕組みのことである。（クラスの代わりに、幾つかのオブジェクト指向言語では、オブジェクトとクラスの機能を兼ね備えた**委譲**と呼ばれる仕組みを使っている。）

(5) **オープンな再帰**。`self`（幾つかの言語では `this`）と呼ばれる特殊変数を通じて、メソッド本体から同一オブジェクトの別のメソッドを呼び出せる機能がある。これは、オブジェクトやクラスを持つほとんどの言語が提供している便利な機能である。`self` の特別な振る舞いは、**遅延束縛**にある。遅延束縛によって、あるクラスで定義されたメソッドが、サブクラスで後に定義される別のメソッドを呼び出せる。

以降の本章の節では、引き続きこれらの機能を説明する。まずはごく単純な「独立した」オブジェクトから始めて、徐々にクラスの強力な形態について考えていく。

後の章では、オブジェクトとクラスの異なる解釈を検討する。第19章では、Java風のオブジェクトとクラスを（他の計算体系で表現するのではなく）直接取り扱う方法を説明する。第27章では、

[†1] Smalltalkのような幾つかのオブジェクト指向言語では、このカプセル化は必須である。つまり、オブジェクトの内部のフィールドは、定義外では単に**名前を指定する**ことができない。他の言語、例えばC++やJavaでは、フィールドに `public` もしくは `private` のどちらを付けることも許されている。反対に、Smalltalkではオブジェクトのすべてのメソッドはどこからでもアクセスできる一方、C++やJavaでは、呼び出し元を同一オブジェクトの他のメソッドに制限するために、**メソッド**に `private` を付けることが許される。ここではそのような改良には立ち入らないが、研究論文（Pierce and Turner [372]; Fisher and Mitchell [154]; Fisher [150]; Fisher and Mitchell [153]; Fisher [151]; Fisher and Reppy [155]）で詳細に考えられている。

ほとんどのオブジェクト指向言語ではカプセル化を不可欠な概念として扱っているが、そうではない言語も幾つもある。CLOS（Bobrow, DeMichiel, Gabriel, Keene, Kiczales, and Moon [47]; Kiczales, des Rivières, and Bobrow [262]）、Cecil（Chambers [94; 95]）、Dylan（Feinberg, Keene, Mathews, and Withington. [145]; Shalit [432]）、およびKEA（Mugridge, Hamer, and Hosking [329]）、そしてCastagna、GhelliおよびLongoの λ-&計算（Castagna, Ghelli, and Longo [93]; Castagna [92]）に見られる**マルチメソッド**は、オブジェクトの状態をメソッドから切り離している。そして特別な型タグを用いて、オーバーロードされた複数のメソッド本体から適切なものをメソッド呼び出し時に選択する。これらの言語においてオブジェクトの生成、メソッド呼び出し、クラス定義などのために内在する機構は、本章で説明する言語とは根本的に異なるが、それらから得られる高水準でのプログラミングイディオムはとても似通っている。

本章で説明する表現に立ち返り、有界量化を用いたクラス生成の実行時効率性を改善する。第 32 章では、純粋関数的な状況で機能する、より野心的な表現を説明する。

18.2 オブジェクト

オブジェクトの最も単純な形態は、何らかの内部**状態**をカプセル化し、**メソッド**の集まりを通じてその状態へのアクセスを提供する、単なるデータ構造である。内部状態は、典型的には、たくさんの書き換え可能な**インスタンス変数**（**フィールド**ともいう）からなる。インスタンス変数はメソッド間では共有されるが、プログラムの他の部分からはアクセスできないようになっている。

本章を通じて扱う題材は、単純なカウンタを表現するオブジェクトである。各カウンタオブジェクトは、一つの数を持ち、二つのメソッドを提供する（つまり、二つのメッセージに応答する）。メソッドの一つはオブジェクトに現在の値を返させる get であり、もう一つは値を増加（increment）させる inc である。

これまでの章で議論した機能でこの振る舞いを実現するなら、オブジェクトの内部状態を参照セルで扱い、メソッドの集まりを関数のレコードで扱うのが単純な方法である。現在の状態が 1 のカウンタオブジェクトは、以下のようになる。

```
  c = let x = ref 1 in
        {get = λ_:Unit. !x,
         inc = λ_:Unit. x:=succ(!x)};
▸ c : {get:Unit→Nat, inc:Unit→Unit}
```

二つのメソッドの本体は両方とも自明な引数（本体で参照する必要がないため、_ で書かれる）を伴った関数として書かれる。この関数抽象は、オブジェクトが作られるときにメソッド本体が評価されないようにすると共に、のちに引数 unit に何度も適用することで、本体を繰り返し評価できるようにする。なお、オブジェクトの状態を、どのようにメソッド間で共有し、プログラムの他の部分からアクセス不可にしているかに注意されたい。状態のカプセル化が変数 x のレキシカルスコープから直接もたらされるのである。

オブジェクト c のメソッドを呼び出すためには、単にレコードのフィールドを取り出し、それを適切な引数に適用すればよい。例えば、以下のようにする。

```
  c.inc unit;
▸ unit : Unit

  c.get unit;
▸ 2 : Nat

  (c.inc unit; c.inc unit; c.get unit);
▸ 4 : Nat
```

メソッド inc が unit を返すので、連続する値の増加を表すのにセミコロン記法（11.3 節）が使える。上記の最終行は、以下のように書いても等価である。

```
  let _ = c.inc unit in let _ = c.inc unit in c.get unit;
```

多くのカウンタを作成し操作したいかもしれないので、この型の短縮名を導入すると便利である。

```
  Counter = {get:Unit→Nat, inc:Unit→Unit};
```

本章で注目するのは、オブジェクトがどのように作られるかであり、より大きなプログラムを組み上げる際にそれがどのように使われるかではない。しかし、オブジェクトを扱う関数を少なくとも

一つ観察し、それが異なる内部表現を持つオブジェクトに対して機能するかを検証したい。次はその些細な例である。カウンタオブジェクトを持ち、その inc メソッドを三回呼び出す関数である。

```
  inc3 = λc:Counter. (c.inc unit; c.inc unit; c.inc unit);
▸ inc3 : Counter → Unit

  (inc3 c; c.get unit);
▸ 7 : Nat
```

18.3 オブジェクト生成器

ここまで、個々のカウンタオブジェクトを一つずつ構築する方法を見てきた。同様の手軽さで、**カウンタ生成器**を書くこともできる。これは、呼び出されるたびに新しいカウンタを作って返す関数である。

```
  newCounter =
    λ_:Unit. let x = ref 1 in
               {get = λ_:Unit. !x,
                inc = λ_:Unit. x:=succ(!x)};
▸ newCounter : Unit → Counter
```

18.4 部分型付け

オブジェクト指向プログラミングが人気である理由の一つは、さまざまな形のオブジェクトが同一のクライアントコードで扱えることにある。例えば、上で定義した Counter オブジェクトに加えて、いかなるときでも初期状態（例えば 1）にリセットするような追加のメソッドを伴うオブジェクトを作るとしよう。

```
  ResetCounter = {get:Unit→Nat, inc:Unit→Unit, reset:Unit→Unit};

  newResetCounter =
    λ_:Unit. let x = ref 1 in
               {get   = λ_:Unit. !x,
                inc   = λ_:Unit. x:=succ(!x),
                reset = λ_:Unit. x:=1};
▸ newResetCounter : Unit → ResetCounter
```

ResetCounter は Counter のフィールドすべて（ともう一つ）を持つので、レコードの部分型付け規則から、ResetCounter <: Counter であるとわかる。これは、カウンタを引数に取る inc3 のような関数は、リセットカウンタに対しても安全に用いることができる、ということを意味する。

```
  rc = newResetCounter unit;
▸ rc : ResetCounter

  (inc3 rc; rc.reset unit; inc3 rc; rc.get unit);
▸ 4 : Nat
```

18.5 インスタンス変数のグループ化

今までのところ、オブジェクトの状態は単一の参照セルで成り立っていた。明らかに、より興味深いオブジェクトは複数のインスタンス変数を持つことが多い。続く節では、複数のインスタンス変数すべてを一つの構成単位として扱えると便利である。これを可能にするために、カウンタの内部表現を参照セルの**レコード**に変更する。そして、メソッド本体におけるインスタンス変数の参照は、このレコードからフィールドを射影することによって行う。

```
c = let r = {x=ref 1} in
      {get = λ_:Unit. !(r.x),
       inc = λ_:Unit. r.x:=succ(!(r.x))};
▶ c : Counter
```

このインスタンス変数レコード（インスタンス変数の値のレコード）の型を、オブジェクトの**表現型**と呼ぶ。

```
CounterRep = {x: Ref Nat};
```

18.6 単純なクラス

newCounter と newResetCounter の定義は、後者に含まれる reset メソッドを除いて同一である。もちろん、どちらの定義も非常に短いので大した差はないのだが、定義がたくさんのページにわたるほど大きくなることは、実用上は容易に起こることである。そのような場合を想像すると、明らかに、共通した機能を一か所に記述する方法が望まれる。その実現のためにほとんどのオブジェクト指向言語で用いられる機構こそ、**クラス**と呼ばれるものである。

現実世界のオブジェクト指向言語におけるクラスの機構は複雑になりがちであり、self、super、可視性の注釈、静的フィールドとメソッド、内部クラス、フレンドクラス、final や Serializable のような注釈などなど、多くの機能が詰め込まれている[†2]。ここではこれらのほとんどを無視して、クラスの最も基本的な側面にのみ着目する。基本的な側面とはつまり、継承によるコードの再利用と、self の遅延束縛である。今ここでは、前者だけを考える。

最も原始的な形では、クラスは単純には、メソッドの集まりを格納しておくデータ構造であり、新たなオブジェクトを生成するために**インスタンス化**されたり、別のクラスを作るために**拡張**されたりする。

リセットカウンタを構築するのに、カウンタ**オブジェクト**のメソッドをただ再利用する、ということはできない。理由は単純であり、どのカウンタオブジェクトにおいても、メソッド本体は、個々のオブジェクトが持つインスタンス変数レコードへの参照を含んでいるからである。明らかに、同じコードを異なるインスタンス変数レコードに再利用できるようにしたい場合、なすべきことは、インスタンス変数に関してメソッドを**抽象化する**ことである。これは上で導入した newCounter 関数を二つの部分に分割することになる。一つは、**任意の**インスタンス変数レコードに対してメソッド本体を定義する部分であり、

[†2] これらの複雑さすべてに寄与する主な原因は、ほとんどのオブジェクト指向言語において、クラスは大規模構造を司る**唯一の**機構だということにある。実際、広く使われている言語のうち、クラスと洗練されたモジュールシステムの両方を提供しているものは、唯一 OCaml だけである。そのため、ほとんどの言語において、クラスは大規模構造に関するあらゆる言語機能のゴミ捨て場になりがちである。

```
counterClass =
  λr:CounterRep.
    {get = λ_:Unit. !(r.x),
     inc = λ_:Unit. r.x:=succ(!(r.x))};
▸ counterClass : CounterRep → Counter
```

もう一つは、インスタンス変数レコードを確保してメソッド本体に供給し、オブジェクトを作る部分である。

```
newCounter =
  λ_:Unit. let r = {x=ref 1} in
             counterClass r;
▸ newCounter : Unit → Counter
```

counterClass のメソッド本体を再利用して、**サブクラス**と呼ばれる、新しいクラスを定義できる。例えば、リセットカウンタのクラスを以下のように定義できる。

```
resetCounterClass =
  λr:CounterRep.
    let super = counterClass r in
      {get   = super.get,
       inc   = super.inc,
       reset = λ_:Unit. r.x:=1};
▸ resetCounterClass : CounterRep → ResetCounter
```

counterClass と同様、この関数は、インスタンス変数レコードを取り、オブジェクトを返す。内部では、まず counterClass を使って**同一の**インスタンス変数レコードを伴うオブジェクトが作られる。この「親オブジェクト」は変数 super に束縛される。次に、super からフィールド get と inc をコピーし、フィールド reset 用に新しい関数を与えることで、新しいオブジェクトが作られる。super は r をもとに作られているため、三つのメソッドすべてが同じインスタンス変数を共有している。

リセットカウンタオブジェクトを構築するためには再び、インスタンス変数のためにメモリを割り当て、実際の処理を行う resetCounterClass を呼び出すだけでよい。

```
newResetCounter =
  λ_:Unit. let r = {x=ref 1} in resetCounterClass r;
▸ newResetCounter : Unit → ResetCounter
```

演習 18.6.1. [推奨, ★★] カウンタに格納されている現在の値から 1 だけ減算する dec という追加のメソッドを伴った、resetCounterClass のサブクラスを書いてみよ。書いたクラスをテストするために、fullref 検査器を利用すること。 □

演習 18.6.2. [★★ ↛] サブクラスのメソッドのレコードの中に、スーパークラスのフィールドのほとんどを明示的にコピーすることは、表記上やや不細工である。サブクラスの中で、スーパークラスのメソッドのコードを明示的に再入力することは避けられるが、依然大量のタイピングを伴う。この書き方で、より大きなオブジェクト指向のプログラムを開発することになったら、すぐに“super with {reset = λ_:Unit. r.x:=1}”（「super と同様のレコードだが、関数 λ_:Unit. r.x:=1 を持つフィールド reset を伴う」という意味）のような、より簡潔な記法が欲しくなるだろう。この記法に対応する、構文、操作的意味論、型付け規則を書き下せ。 □

これらのクラスは**値**であって、**型**ではないことを重視すべきである。望むなら、ちょうど同じ型

を持つオブジェクトを生成する多くのクラスを書くこともできる。C++ や Java といった主流のオブジェクト指向言語では、クラスの位置付けはより複雑である。つまり、コンパイル時に型として使われ、実行時にデータ構造として使われる。この点については、19.3 節でさらに議論する。

18.7 インスタンス変数の付加

カウンタオブジェクトとリセットカウンタオブジェクトは、たまたま全く同一の内部表現を利用していた。しかし、一般的には、サブクラスは派生元のスーパークラスから、メソッドのみならずインスタンス変数も拡張する必要がありうる。例えば、「バックアップカウンタ」というクラスを定義したいとしよう。このクラスの reset メソッドは、状態を、定数値にリセットする代わりに、最後にメソッド backup が呼ばれたとき保持していた値にリセットする。

```
BackupCounter = {get:Unit→Nat, inc:Unit→Unit,
                 reset:Unit→Unit, backup: Unit→Unit};
```

バックアップカウンタを実装するためには、バックアップされた状態の値を保存しておくための、追加のインスタンス変数が必要である。

```
BackupCounterRep = {x: Ref Nat, b: Ref Nat};
```

counterClass から resetCounterClass を派生させたときは、get と inc のメソッドをコピーして、reset を加えた。それと全く同様に、resetCounterClass から backupCounterClass を派生させるには、get と inc のメソッドをコピーして、reset と backup を新たに与える。

```
backupCounterClass =
  λr:BackupCounterRep.
    let super = resetCounterClass r in
      {get    = super.get,
       inc    = super.inc,
       reset  = λ_:Unit. r.x:=!(r.b),
       backup = λ_:Unit. r.b:=!(r.x)};
▸ backupCounterClass : BackupCounterRep → BackupCounter
```

この定義について、興味深いことが二つある。一つ目は、親オブジェクト super が reset メソッドを含んでいるが、異なる振る舞いを得るために新たな実装を書くようにした点である。新しいクラスはスーパークラスの reset メソッドを**オーバーライド**している。二つ目は、super を構築する式を型付けするために、部分型付けがここでは本質的に使われている点である。つまり、resetCounterClass は、引数 r の実際の型 BackupCounterRep の上位型である、CounterRep 型の引数を期待している。言い換えると、実際には、メソッドが要求するより大きなインスタンス変数レコードを親オブジェクトに提供している。

演習 18.7.1. [推奨, ★★] 二つ目の「バックアップレジスタ」を制御する二つのメソッド reset2 と backup2 を伴った、backupCounterClass の部分型を定義せよ。このレジスタは backupCounterClass で加えられたバックアップレジスタとは完全に独立で、reset 呼び出しは最後に backup が呼ばれたときの値に復元し（これは現在の動作）、reset2 呼び出しは最後に backup2 が呼ばれたときの値に復元するようにせよ。作成したクラスをテストするのに fullref 検査器を利用せよ。 □

18.8 スーパークラスのメソッド呼び出し

ここまでのところ、変数 super はスーパークラスから新たなサブクラスに機能をコピーするときに使われてきた。super はまた、メソッド定義の本体中で、スーパークラスの振る舞いを何か追加のもので**拡張**するときにも使える。例えば、backupCounter クラスの変種として、各 inc メソッドの呼び出しの前に backup が自動的に呼ばれるものが欲しいとしよう。（なぜそんなクラスが有用かは誰も知らない。ただの例である。）

```
funnyBackupCounterClass =
  λr:BackupCounterRep.
    let super = backupCounterClass r in
       {get = super.get,
        inc = λ_:Unit. (super.backup unit; super.inc unit),
        reset = super.reset,
        backup = super.backup};
▸ funnyBackupCounterClass : BackupCounterRep → BackupCounter
```

inc の新しい定義で super.inc と super.backup を呼び出すことで、スーパークラスでの inc や backup のコードを繰り返さなくてもよくなっていることに注意されたい。より大きな例では、このような状況で機能の重複を大幅に節約できることがある。

18.9 self を伴ったクラス

最後の拡張は、クラスのメソッド同士の相互参照を self を通して許すことである。この拡張の動機がわかる例として、現在のカウンタ値を特定の数にセットするために、外から利用できる set メソッドのある、カウンタクラスを実装したいとしよう。

```
SetCounter = {get:Unit→Nat, set:Nat→Unit, inc:Unit→Unit};
```

さらに、直接インスタンス変数 x を読んだり破壊的に代入したりするのではなく、set と get を用いる inc メソッドが欲しいとしよう。（set と get の定義がそれぞれ多くのページを費やすような、もっと大きな例を想像してみよ。）get、set、inc はすべて同じクラス内で定義されるので、ここで求められるのは、このクラスのメソッドが本質的に相互再帰するようにすることである。

11.11 節で、fix 演算子を使って相互再帰的関数のレコードを構築する方法を扱った。メソッドのレコードを、自身も関数のレコードであるような引数（self と呼ぶ）を使って単純に抽象化する。そして fix 演算子を用いることで「結び目を縛り」、まさしく我々が構築しているレコードが self として渡されるそのものとなるようにする。

```
setCounterClass =
  λr:CounterRep.
    fix
      (λself: SetCounter.
        {get = λ_:Unit. !(r.x),
         set = λi:Nat.  r.x:=i,
         inc = λ_:Unit. self.set (succ (self.get unit))});
▸ setCounterClass : CounterRep → SetCounter
```

このクラスは、親クラスを持たない。したがって、変数 super は必要ない。その代わり inc メソッドの本体は、self として渡されたメソッドのレコードから get を呼び出し、続いて set を呼び出す。fix の利用は、完全に setCounterClass の内部で閉じている。そしてセットカウンタはこれまでと全く同じように作られる。

```
newSetCounter =
  λ_:Unit. let r = {x=ref 1} in
             setCounterClass r;
▸ newSetCounter : Unit → SetCounter
```

18.10 selfを介したオープンな再帰

ほとんどのオブジェクト指向言語では、**オープンな再帰**や self の**遅延束縛**として知られる、より一般的な形でのメソッド間の再帰呼び出しを実際にサポートしている。クラス定義から、fix の使用を取り除くことで、このより一般的な振る舞いを達成できる。

```
setCounterClass =
  λr:CounterRep.
     λself: SetCounter.
       {get = λ_:Unit. !(r.x),
        set = λi:Nat.  r.x:=i,
        inc = λ_:Unit. self.set (succ (self.get unit))};
▸ setCounterClass : CounterRep → SetCounter → SetCounter
```

そして、代わりにオブジェクト生成関数の中で fix を用いる。

```
newSetCounter =
  λ_:Unit. let r = {x=ref 1} in
             fix (setCounterClass r);
▸ newSetCounter : Unit → SetCounter
```

fix の使用箇所を動かすことで setCounterClass の型が変わることに注意されたい。つまり、インスタンス変数レコードに対して抽象化されるだけでなく、「self オブジェクト」に対しても抽象化される。インスタンス化の際にはインスタンス変数レコードと self オブジェクトの両方を与える。

self を介したオープンな再帰がなぜ興味深いかというと、**スーパークラス**のメソッドが**サブクラス**のメソッドを、スーパークラスが定義されるときにサブクラスが存在しなくても呼び出せるからである。実質的には、self の解釈が変化して、「このクラスのメソッド」ではなく、「現在のオブジェクトがインスタンス化しているクラス（このクラスのサブクラスかもしれない）のメソッド」へのアクセスを提供する。

例えば、set メソッドが何回呼び出されたかを保存する、セットカウンタのサブクラスを構築したいとしよう。このようなカウンタをここでは計器付きカウンタ（instrumented counter）と呼ぶ。このクラスのインターフェイスは、アクセス数を取り出す追加の操作を一つ含む。

```
InstrCounter = {get:Unit→Nat, set:Nat→Unit,
                inc:Unit→Unit, accesses:Unit→Nat};
```

そして、表現は、アクセス数を表すインスタンス変数を一つ含む。

```
InstrCounterRep = {x: Ref Nat, a: Ref Nat};
```

計器付きカウンタクラスの定義において、inc と get メソッドは、上で定義した setCounterClass からコピーされる。accesses メソッドは普通の方法で書かれる。set メソッドでは、まずアクセス数を一つ増やし、次に super を使ってスーパークラスの set を呼び出す。

```
instrCounterClass =
  λr:InstrCounterRep.
     λself: InstrCounter.
       let super = setCounterClass r self in
         {get = super.get,
          set = λi:Nat. (r.a:=succ(!(r.a)); super.set i),
          inc = super.inc,
          accesses = λ_:Unit. !(r.a)};
▸ instrCounterClass : InstrCounterRep →
                        InstrCounter → InstrCounter
```

self を介したオープンな再帰のために、inc 本体からの set の呼び出しでインスタンス変数 a に 1 が加算される。しかしここで、set の加算するという振る舞いはサブクラスで定義され、inc はスーパークラスで定義されている。

18.11 オープンな再帰と評価順序

instrCounterClass の定義には一つ問題がある。インスタンスを構築するためには使えないのである。newInstrCounter を普通の方法で書いたとすると、

```
newInstrCounter =
  λ_:Unit. let r = {x=ref 1, a=ref 0} in
               fix (instrCounterClass r);
▸ newInstrCounter : Unit → InstrCounter
```

となる。次に unit に適用し、計器付きカウンタを作ろうとしたとすると、

```
ic = newInstrCounter unit;
```

評価器は発散する。これがどのように起きるかを見るために、この項から始めたときの、一連の評価ステップを考える。

(1) まず unit に対して newInstrCounter を適用すると、

```
let r = {x=ref 1, a=ref 0} in fix (instrCounterClass r)
```

を得る。

(2) 次に、二つの ref セルを確保してレコード（<ivars> と呼ぶことにする）の中に格納し、残りの部分では r を <ivars> で置き換える。

```
fix (instrCounterClass <ivars>)
```

(3) <ivars> を instrCounterClass に渡す。instrCounterClass は二つのラムダ抽象で始まっているので、self を受け取る関数が即座に得られる。ここで <imethods> は、計器付きカウンタメソッドのレコードである。

```
fix (λself:InstrCounter.
         let super = setCounterClass <ivars> self in <imethods>)
```

この関数を <f> と呼び、現在の状態を (fix <f>) と書く。

(4) fix に対して、評価規則（110 ページ図 11-12 の E-FIX）を適用する。この規則は <f> の本体中の self を (fix <f>) に置換することで fix <f> を「展開」し、次を得る。

```
let super = setCounterClass <ivars> (fix <f>) in <imethods>
```

(5) ここで、setCounterClass の <ivars> に対する適用を簡約し、次を得る。

```
let super = (λself:SetCounter. <smethods>) (fix <f>)
            in <imethods>
```

ただし `<smethods>` はセットカウンタメソッドのレコードである。

(6) 関数適用に関する評価規則により、`(λself:SetCounter. <smethods>)` の `(fix <f>)` への適用は、後者が値に簡約されるまで簡約できない。そこで、評価の次のステップでは、再度 `fix <f>` を展開し、次を得る。

```
let super = (λself:SetCounter. <smethods>)
               (let super = setCounterClass <ivars> (fix <f>)
                in <imethods>)
in <imethods>
```

(7) 外側のラムダ抽象に与えられた引数がまだ値になっていないので、内側の評価を続けなければならない。`setCounterClass` の `<ivars>` に対して適用を行い、次を得る。

```
let super = (λself:SetCounter. <smethods>)
               (let super = (λself:SetCounter. <smethods>)
                               (fix <f>)
                in <imethods>)
in <imethods>
```

(8) 外側の関数適用と同様の形の関数適用が内側にも生成された。以前と同じように、この内側の適用はその引数 `fix <f>` が完全に評価されるまでは簡約されない。したがって、次のステップは `fix <f>` を再度展開し、ステップ (6) の中と同じ形の、さらに深くネストされた式が得られる。この時点で、外側の適用を評価する段階に**絶対**辿り着か**ない**ことは明らかである。

直観的には、ここでの問題は、`fix` 演算子に対する引数がその引数 `self` を使うのが早すぎることにある。`fix` の操作的意味論がその定義上期待されていることは、`fix` を何らかの関数 `λx.t` に適用するとき、本体 `t` は `x` を「保護された」位置でのみ参照するべきだ、ということである。例えば、109 ページの `iseven` は、`fix` を `λie. λx. ...` という形式の関数に適用することで定義した。ここで、本体における `ie` の再帰的な参照は、`x` に対する抽象化によって保護されている。対照的に、`instrCounterClass` の定義は、`super` の値を計算するときに `self` をすぐに使おうとしている。

ここから先に進むには幾つかの選択肢がある。

- `self` への参照が評価されるのが早すぎるのを防ぐため、例えばダミーのラムダ抽象によって、`instrCounterClass` の中の `self` への参照を保護できる。本節ではこの解法を説明する。完全に満足できるものではないが、今まで見た機構を用いて記述、理解するのが容易であることがわかるだろう。同時に便利でもあることは、後ほど第 32 章で、純粋関数的なオブジェクトの表現を考えるときにわかるだろう。
- クラスの意味論をモデル化するために、低水準の言語機能を利用する異なる方法を探すこともできる。例えば、クラスのメソッドテーブルを構築するために `fix` を用いず、代わりに、参照を利用してもっと明示的に構築できる。このアイディアは 18.12 節で説明して、第 27 章でより洗練させる。
- オブジェクトとクラスを、ラムダ抽象、レコード、`fix` を用いて表現するのをやめることもできる。その代わりに、オブジェクトとクラスを言語プリミティブとして扱い、それらの評価（と型付けの）規則を与える。すると、オブジェクトとクラスの意図した振る舞いに合う評価規則を単純に選択できるようになり、関数適用と `fix` に与えられた規則にからむ問題で回避策を講じる必要がなくなる。この方法は、第 19 章で説明する。

評価順序を制御するためにダミーのラムダ抽象を使うことは、関数的プログラミングのコミュニティではよく知られたトリックである。そのアイディアは、任意の表現 t を関数 λ_:Unit.t （この関数を**サンク**と呼ぶ）に変換できるというものである。この t の「サンク形式」は構文上の値であり、t の評価に関するすべての計算は、このサンクが unit に適用されるまで延期される。これは、t を未評価のまま受け渡し、後でその結果を求めるという方法を提供してくれる。

当面やりたいことは、self の評価を遅らせることである。それは、self の型をオブジェクト（例えば SetCounter）からオブジェクトのサンク（Unit→SetCounter）に変えることで可能である。それには、(1) クラスの引数 self の型を変え、(2) 結果オブジェクトを作る前にダミーの抽象を付け加え、(3) メソッド本体に現れるすべての self を (self unit) へ変える必要がある。

```
setCounterClass =
  λr:CounterRep.
  λself: Unit→SetCounter.
    λ_:Unit.
      {get = λ_:Unit. !(r.x),
       set = λi:Nat.  r.x:=i,
       inc = λ_:Unit. (self unit).set(succ((self unit).get unit))};
▸ setCounterClass : CounterRep →
                     (Unit→SetCounter) → Unit → SetCounter
```

newSetCounter の型は変えたくない（newSetCounter はやはりオブジェクトを返すべきである）ので、setCounterClass の不動点を取って得られたサンクに unit 引数を渡すよう、定義を少し変更する必要もある。

```
newSetCounter =
  λ_:Unit. let r = {x=ref 1} in
             fix (setCounterClass r) unit;
▸ newSetCounter : Unit → SetCounter
```

似たような変更は、instrCounterClass の定義でも必要である。注意として、これらの変更は、実際には判断を少しも必要としない。一度 self の型を変えてしまえば、他の変更は型付け規則から決定されるからである。

```
instrCounterClass =
  λr:InstrCounterRep.
  λself: Unit→InstrCounter.
    λ_:Unit.
      let super = setCounterClass r self unit in
         {get = super.get,
          set = λi:Nat. (r.a:=succ(!(r.a)); super.set i),
          inc = super.inc,
          accesses = λ_:Unit. !(r.a)};
▸ instrCounterClass : InstrCounterRep →
                       (Unit→InstrCounter) → Unit → InstrCounter
```

最後に、fix で生成されるサンクへダミーの引数が供給されるように newInstrCounter を変更する。

```
newInstrCounter =
  λ_:Unit. let r = {x=ref 1, a=ref 0} in
             fix (instrCounterClass r) unit;
▸ newInstrCounter : Unit → InstrCounter
```

ここに至って、実際にオブジェクトを作るために newInstrCounter を使える。

```
  ic = newInstrCounter unit;
▶ ic : InstrCounter
```

これは、サンクを加える前には発散してしまうステップだったことを思い出そう。

以下のテストでは、accesses メソッドが set と inc 両方の呼び出し回数をどう数えているかがわかる。これは、意図した通りの動作である。

```
  (ic.set 5; ic.accesses unit);
▶ 1 : Nat

  (ic.inc unit; ic.get unit);
▶ 6 : Nat

  ic.accesses unit;
▶ 2 : Nat
```

演習 18.11.1. [推奨, ★★★]　fullref 検査器を用いて、上のクラスに対して以下の拡張を実装せよ。

(1) instrCounterClass を書き直し、get の呼び出しをも数えられるようにせよ。
(2) 変更した instrCounterClass をサブクラスで拡張し、180 ページで行ったように、reset メソッドを加えよ。
(3) それとは別のサブクラスを付け加え、18.7 節で行ったようにバックアップ機能を提供せよ。

□

18.12 より効率的な実装

前述のテストは、我々のクラスの実現方法が、Smalltalk や C++、Java といった言語における self を介したメソッド呼び出しの「オープンな再帰」の振る舞いに合っていることを明らかにしている。しかし、効率性の観点からは、その方法が完全には満足できるものでないことは注意すべきである。fix の計算を収束させるために挿入したすべてのサンクは、クラスのメソッドテーブルの計算を延期させる効果があった。特に、メソッド本体での self 呼び出しのすべてが (self unit) になったことに注意されたい。つまり、self のメソッドは、その中の一つへの再帰呼び出しが起こるたびごとに、その場で再計算されるのである。

オブジェクトを作るときにクラス階層において「結び目を縛る」のに、不動点ではなく参照セルを用いれば、これらすべての再計算を避けることができる[†3]。先ほどは、self と呼ばれるメソッドのレコードでクラスを抽象化し、その後で fix を使って self を構築した。その代わり、メソッドのレコードへの**参照**でクラスを抽象化することにし、レコードは**最初に**確保するようにする。つまり、クラスのメソッドのための（ダミーの値で初期化された）ヒープセルをまず確保することでクラスをインスタンス化し、次に本当のメソッドを（ヒープセルに対するポインタを渡すことで）構築する（渡したポインタは本当のメソッドの中で再帰呼び出しに用いることができる）。そして最終的に本当のメソッドを保持するようヒープセルを修正する。例として、setCounterClass を再び考える。

[†3] これは、本質的には演習 13.5.8 に対する解法の中で使ったアイディアと同じである。そのアイディアを、Source 型の共変性を有効に使うことでクラスの構築にも適用できる、という洞察をくれた James Riely に感謝する。

```
setCounterClass =
  λr:CounterRep. λself: Ref SetCounter.
    {get = λ_:Unit. !(r.x),
     set = λi:Nat. r.x:=i,
     inc = λ_:Unit. (!self).set (succ ((!self).get unit))};
▸ setCounterClass : CounterRep → (Ref SetCounter) → SetCounter
```

この引数 self は、現在考えているオブジェクトのメソッドがある領域へのポインタである。setCounterClass が呼ばれる時点では、この領域はダミーの値で初期化されている。

```
dummySetCounter =
  {get = λ_:Unit. 0,
   set = λi:Nat.  unit,
   inc = λ_:Unit. unit};
▸ dummySetCounter : SetCounter

newSetCounter =
  λ_:Unit.
    let r = {x=ref 1} in
    let cAux = ref dummySetCounter in
    (cAux := (setCounterClass r cAux); !cAux);
▸ newSetCounter : Unit → SetCounter
```

しかし、すべての参照先の値を取り出す操作（!self）がラムダ抽象により保護されているので、newSetCounter により修正されるまでは、実際に領域から値が取り出されることはない。

setCounterClass のサブクラスの構築を実現するには、型をさらに改良する必要がある。各クラスは引数 self が、クラスが構築するメソッドのレコードと同じ型を持つことを期待する。つまり、計器付きカウンタのサブクラスを定義する場合、そのクラスの引数 self は、計器付きカウンタメソッドのレコードへのポインタになる。しかし、15.5 節で扱ったように、Ref SetCounter 型と Ref InstrCounter 型は互換性がない。つまり、後者を前者に昇格するのは健全ではない。それは、instrCounterClass の定義の中で super を作ろうとするときに、問題（つまり、引数型の不一致）を引き起こす。

```
instrCounterClass =
  λr:InstrCounterRep. λself: Ref InstrCounter.
  let super = setCounterClass r self in
  {get = super.get,
   set = λi:Nat. (r.a:=succ(!(r.a)); super.set i),
   inc = super.inc,
   accesses = λ_:Unit. !(r.a)};
▸ Error: parameter type mismatch
```

この困難から逃れる方法は、self の型における Ref 構築子を Source 構築子に置き換えることである。つまり、メソッドポインタから読む機能だけをクラスへ渡し、メソッドポインタへ書き込む機能は渡さない（どのみち必要ないのである）。15.5 節で見たように、Source は共変の部分型付けを許す。つまり、Source InstrCounter <: Source SetCounter が成り立ち、instrCounterClass 中の super 生成は正しく型付けされるようになる。

```
setCounterClass =
  λr:CounterRep. λself: Source SetCounter.
    {get = λ_:Unit. !(r.x),
     set = λi:Nat. r.x:=i,
     inc = λ_:Unit. (!self).set (succ ((!self).get unit))};
▸ setCounterClass : CounterRep → (Source SetCounter) → SetCounter

instrCounterClass =
  λr:InstrCounterRep. λself: Source InstrCounter.
  let super = setCounterClass r self in
  {get = super.get,
   set = λi:Nat. (r.a:=succ(!(r.a)); super.set i),
   inc = super.inc,
   accesses = λ_:Unit. !(r.a)};
▸ instrCounterClass : InstrCounterRep →
                        (Source InstrCounter) → InstrCounter
```

計器付きカウンタオブジェクトを構築するためには、先ほどと同様、まず、self ポインタの初期値を与えるための、計器付きカウンタメソッドのダミーの集まりを定義する。

```
dummyInstrCounter =
  {get = λ_:Unit. 0,
   set = λi:Nat.  unit,
   inc = λ_:Unit. unit,
   accesses = λ_:Unit. 0};
▸ dummyInstrCounter : InstrCounter
```

そして、インスタンス変数とメソッドのためのヒープ空間を確保することでオブジェクトを生成する。その後、instrCounterClass を呼んで実際のメソッドを構築し、最後に参照セルを修正する。

```
newInstrCounter =
  λ_:Unit.
    let r = {x=ref 1, a=ref 0} in
    let cAux = ref dummyInstrCounter in
    (cAux := (instrCounterClass r cAux); !cAux);
▸ newInstrCounter : Unit → InstrCounter
```

メソッドテーブルを構築するための（instrCounterClass と counterClass の中の）コードは、メソッド呼び出しごとに毎回ではなく、オブジェクト生成時に一度だけ呼ばれるようになる。これは、目指したことを達成するが、まだ望んでいたほど効率的でない。結局、計器付きカウンタオブジェクトそれぞれに対して構築するメソッドテーブルは常に全く同一であるから、クラスを定義するときにメソッドテーブルの計算を一度きり行い、二度と再計算しないようにできるべきだと思われる。第 27 章において、第 26 章で導入する**有界量化**を用いてこれがどのように実現できるかを見る。

18.13 要点のまとめ

本章の最初の節では、オブジェクト指向プログラミングの特徴的な機能を幾つか挙げた。それらの機能を思い出し、本章で説明した例とどのように関係があるか簡潔に議論しよう。

(1) **複数の表現**。本章で扱ったオブジェクトのすべてはカウンタである。つまり型 Counter に属している。しかし、その表現はさまざまであって、18.2 節の参照セルから、18.9 節で扱っ

た幾つかの参照を持つレコードまで幅広い。各オブジェクトは関数のレコードで、自身の内部表現に適した Counter メソッド（とおそらく他のメソッドも）の実装を提供している。

(2) **カプセル化**。オブジェクトのインスタンス変数はそのオブジェクトのメソッドのみからアクセスできるという事実は、オブジェクトを構成した方法から直接導かれる。インスタンス変数レコードを構築子関数に渡すことによってメソッドを構築するからである。インスタンス変数はメソッドの内部からのみ**名前を指定できる**のは明らかである。

(3) **部分型付け**。本章のやり方に従えば、オブジェクトの型の間にある部分型関係は、関数のレコードの型の間にある、通常の部分型関係になる。

(4) **継承**。既に存在するスーパークラスから、新しく定義されるサブクラスに、メソッドの実装をコピーすることで継承をモデル化した。そこには、幾つかの興味深い専門的な事柄がある。厳密なことを言うと、スーパークラスと新しいサブクラスは両方とも、インスタンス変数からメソッドのレコードへの**関数**である。サブクラスはインスタンス変数を受け取る。そして、与えられたインスタンス変数でスーパークラスをインスタンス化し、同じ変数への操作を行うスーパークラスのメソッドのレコードを構成する。

(5) **オープンな再帰**。実世界のオブジェクト指向言語で self（や this）によって導入されるオープンな再帰は、ここでは、クラスをインスタンス変数に対してだけではなく、引数 self に対しても抽象化することでモデル化した。引数 self は、メソッド中で同じオブジェクトの他のメソッドを参照するために使える。この引数が持つ値はオブジェクト生成時に、fix を用いて「結び目を縛る」ことで構築される。

演習 18.13.1. [★★★] ある目的のために有用な、オブジェクトのもう一つの機能は、**オブジェクト同一性**の概念である。つまり、二つの引数が全く同一のオブジェクトに評価されるときに true を返し、（new 関数の異なる呼び出しにより）異なる時点で生成されたオブジェクトを評価したときには false を返すような sameObject の操作である。本章のオブジェクトのモデルは、どのように拡張すればオブジェクト同一性を実現できるだろうか。 □

18.14 注記

オブジェクトの表現方法は、プログラミング言語研究コミュニティに例や問題を提供する、主要な供給源である。初期の表現方法は Reynolds [410] によって与えられた。この分野で一般的な興味の対象となったのは Cardelli の論文 [73] が端緒である。不動点の観点からの self の理解は、Cook [113]、Cook and Palsberg [112]、Kamin [248]、および Reddy [397] によって開拓された。それらのモデルの間の関係は、Kamin and Reddy [249] および Bruce [58] により探究された。

この分野における初期の重要な論文の多くは Gunter and Mitchell [189] にまとめられている。より最近の研究内容は Brucem, Cardelli and Pierce [63] および Abadi and Cardelli [4] にてサーベイされている。Bruce [60] では、この分野における最新の進展の様子が描かれている。オブジェクトとその型システムに対する、別の基礎的なアプローチは Palsberg and Schwartzbach [350] と Castagna [92] にある。

歴史に関してさらなる注記が幾つか第 32 章の終わりにもある。

継承は非常に過大評価されている。 —*Grady Booch*

第19章

事例：Featherweight Java

本章の例は Featherweight Java の項である（図 19-1 から図 19-4 まで）。対応する OCaml 実装はない。それよりは、FJ は Java の厳密な部分集合として設計されているから、本章の例を実行するには任意の Java 実装を用いることができる。

第 18 章では、オブジェクト指向プログラミングの主要な機能を、部分型付けとレコード、参照を持つラムダ計算を用いてモデル化する方法を説明した。その目的は、それらの機能をより基本的な機能で**表現**することで、理解を深めることであった。本章では異なるアプローチをとる。前章のアイディアを適合させて、Java に基づいた核となるオブジェクト指向言語を**直接**扱う方法を示す。Java の知識を前提とする。

19.1 導入

プログラミング言語のような現実世界の複雑な人工物を設計するには、形式的にモデル化することが大きな手助けとなる。形式モデルを使うと、設計のある側面を正確に説明したり、その性質を記述・証明したり、そこまでしないと見落としかねない問題に注意を向けたりできる。しかしモデルを形式化する際には、完全さとコンパクトさの間で葛藤がある。モデルが同時により多くの側面を扱おうとするほど、モデルは大きく不格好になる。往々にして、完全さでは劣ってもよりコンパクトなモデルを選ぶほうが賢明なこともある。最大の知見を最小の投資で得られるからである。この戦略は、Java の形式的な性質に関する近年の論文で立て続けに採用されている。つまり、並列性やリフレクションなどの発展的な機能を省略する代わりに、言語全体の中でも、よく理解された理論が適用できる部分に集中するということである。

Featherweight Java（FJ とも呼ぶ）は、Igarashi, Pierce, and Wadler [231] が提案したもので、Java の型システムをモデル化するための**最小限の**核となる計算体系の候補である。FJ の設計では、ほとんど執拗といえるくらい、完全さよりコンパクトさを重視している。項の形式には、オブジェクト生成、メソッド呼び出し、フィールド参照、キャスト、変数の、たった五つしかない。文法、型付け規則、操作的意味論は、（レターサイズの）1 ページにゆうに収まる。実際その設計の目標は、できる限り多くの機能を省略する（破壊的代入すら省略している）一方、Java の型付けの中心的な機能を維持することであった。すべての FJ プログラムは、そのまま実行可能な Java プログラムであるという点において、FJ と、Java の純粋関数的な中心部分との間には、直接的な対応関係がある。

FJ は、ラムダ計算や Abadi と Cardelli のオブジェクト計算 [4] より少しだけ大きく、Java のようなクラスベースの言語の他の形式モデルである Drossopoulou, Eisenbach, and Khurshid [140]、Syme [451]、Nipkow and von Oheimb [339]、Flatt, Krishnamurthi, and Felleisen [158; 159] などよりずっと小さい。この小ささのおかげで、FJ ではごくわずかの重要な問題に焦点を絞れる。例えば、小ステップ操作的意味論における Java のキャスト構文の挙動を捉えるのは想像よりややこしいことを後に見る。

FJ の主な応用は、Java の拡張をモデル化することである。FJ 自身は非常にコンパクトなので、拡張の本質的な側面に集中して考えられる。さらに、純粋な FJ での型安全性の証明は非常に簡単なので、大幅な拡張に対する厳密な安全性の証明も扱える範囲に収まりやすい。FJ の最初の論文では、GJ（Bracha, Odersky, Stoutamire, and Wadler [54]）風の総称クラス・メソッドで FJ を拡張し、そのような実用性を示した。続く論文（Igarashi, Pierce, and Wadler [233]）では、Java プログラム

から GJ への移行を簡単にするために GJ に導入された機能である **raw 型**[†1]を形式化した。Igarashi and Pierce [232] では FJ を Java の**内部クラス**の機能を研究する土台として使った。型を保存するコンパイル（League, Trifonov, and Shao [275]）や Java の意味論の基礎（Studer [448]）の研究でも FJ は使われている。

FJ の設計の目標は、型安全性の証明をできる限り簡潔にしつつも、完全な Java の中心的な機能に関する安全性の議論の本質を捉えたままにすることであった。安全性の証明を**長く**するが、大きく**異なる**ものにするわけではない言語機能は、何であれ省略の候補だった。この種の他の研究と同じように、FJ は並行性やリフレクションなどの発展的な機能を省略している。FJ が省略した Java の他の機能には、破壊的代入、インターフェイス、オーバーロード、super へのメッセージ、null ポインタ、基本型（int、bool など）、抽象メソッド定義、内部クラス、サブクラスのフィールドでスーパークラスのフィールドを隠すこと、アクセス制御（public、private など）、例外などがある。FJ がモデル化**する** Java の機能は、相互再帰的なクラス定義、オブジェクト生成、フィールド参照、メソッド呼び出し、メソッドオーバーライド、this を介したメソッド再帰、部分型付け、キャストなどである。

FJ の鍵となる単純化は、破壊的代入の省略である。オブジェクトのフィールドはコンストラクタで初期化され、それ以降に変えられることはないと仮定している。これによって FJ は、Java の「関数的な」サブセットに制限される。これでは、多くの一般的な Java のイディオムは表現できない。それにもかかわらず、このサブセットは計算的に完全であり（ラムダ計算を FJ で表現するのは容易である）、有用なプログラムを書ける程度には、十分な大きさがある。例えば、Felleisen と Friedman の Java の教科書 [146] に載っている多くのプログラムは、純粋関数的なスタイルを用いている。

19.2 概観

FJ においてプログラムは、クラス定義の集まりと、一つの評価される項（完全な Java における main メソッドの本体に相当する）からなる。以下に FJ での典型的なクラス定義を示す。

```
class A extends Object { A() { super(); } }

class B extends Object { B() { super(); } }

class Pair extends Object {
  Object fst;
  Object snd;
  // Constructor:
  Pair(Object fst, Object snd) {
    super(); this.fst=fst; this.snd=snd; }
  // Method definition:
  Pair setfst(Object newfst) {
    return new Pair(newfst, this.snd); } }
```

文法を規則的にするため、スーパークラスは常に書き（Object の場合でも）、コンストラクタも常に書き（クラス A や B のように自明でも）、フィールド参照やメソッド呼び出しのレシーバも、レシーバが this の場合であっても（this.snd のように）常に書く。コンストラクタは常に同じ定型化した形式を取る。すなわち、フィールドと同じ名前の引数をフィールドの数だけ受け取り、スーパークラスのフィールドを初期化するため super コンストラクタを呼び、そして、残りのフィール

[†1]（訳注）村上雅章 訳『Java 言語仕様第 3 版』（ピアソン・エデュケーション、2006 年）では「未加工型」と呼ばれている。

ドを対応する引数で初期化する。今回の例では、三つのクラスのスーパークラスはどれも Object である。Object はフィールドを持たないので、super の呼び出しに引数はない。コンストラクタは FJ のプログラムの中で super と = が現れる唯一の場所である。FJ は副作用のある操作を提供しないので、メソッドの本体は setfst() のように、常に return とそれに続く一つの項だけとなる。

FJ の項には五つの形式がある。先の例の中では、new A()、new B()、new Pair(...,...) が**オブジェクトコンストラクタ**であり、....setfst(...) が**メソッド呼び出し**である。setfst の本体では、項 this.snd が**フィールド参照**であり、newfst や this の出現は**変数**である[†2]。上記の定義の文脈で、項

```
new Pair(new A(), new B()).setfst(new B())
```

は new Pair(new B(), new B()) へと評価される。

残る項の形式は**キャスト**（151 ページを見よ）である。項

```
((Pair)new Pair(new Pair(new A(), new B()),
                new A()).fst).snd
```

は、new B() と評価される。部分項 (Pair)t はキャストである。ただし t は new Pair(...).fst である。t は fst のフィールド参照であり、fst は Object を持つと宣言されている。一方その次の snd のフィールド参照は、Pair 上でしか有効でない。したがって、ここでキャストが必要になっている。評価規則は、fst フィールドに保存されている Object が Pair であることを実行時に検査する（この例では検査は成功する）。

副作用を捨てると嬉しい副作用がある。完全に FJ の文法だけで、評価を簡単に形式化できるようになるのである。ヒープをモデル化するための余分な仕組みは必要ない（第 13 章を参照）。FJ には三つの基本的な計算規則がある。フィールド参照と、メソッド呼び出しと、キャストである。ラムダ計算では、関数適用の評価規則は、関数はまずラムダ抽象に簡約されると仮定していることを思い出そう。同じように FJ の評価規則は、処理にかかわるオブジェクトはまず new 項へ簡約されることを仮定している。ラムダ計算のスローガンは「すべてのものは関数だ」だったが、FJ では、「すべてのものはオブジェクトだ」となる。

次はフィールド参照の規則（E-ProjNew）の動作例である。

$$\texttt{new Pair(new A(), new B()).snd} \quad \longrightarrow \quad \texttt{new B()}$$

オブジェクトコンストラクタの文法は定型化されているので、このコンストラクタが各フィールドに対して一つの引数を、フィールドの宣言順に持つことがわかっている。ここではフィールドは fst と snd である。snd フィールドの参照は二番目の引数を選択する。

以下はメソッド呼び出しの規則（E-InvkNew）の動作例である。

$$\begin{array}{l} \texttt{new Pair(new A(), new B()).setfst(new B())} \\ \longrightarrow \left[\begin{array}{l} \texttt{newfst} \mapsto \texttt{new B(),} \\ \texttt{this} \mapsto \texttt{new Pair(new A(),new B())} \end{array}\right] \\ \quad \texttt{new Pair(newfst, this.snd)} \\ \quad \text{つまり、}\texttt{new Pair(new B(), new Pair(new A(), new B()).snd)} \end{array}$$

この呼び出しのレシーバはオブジェクト new Pair(new A(),new B()) なので、Pair クラスの setfst メソッドを調べる。その結果、そのメソッドは仮引数 newfst と本体 new Pair(newfst, this.snd) を持つことがわかる。この呼び出しは、メソッド本体の仮引数を実引

[†2] this をキーワードでなく変数として扱うという点で、FJ の文法は Java と若干異なる。

数に置き換え、かつ特殊変数 this をレシーバオブジェクトに置き換えたものに簡約される。これはラムダ計算のベータ簡約の規則（E-AppAbs）と似ている。主な違いは、メソッド本体を探すべき場所をレシーバのクラスが決定すること（メソッドオーバーライドを実現するため）と、レシーバを this に代入すること（「self を介したオープンな再帰」を実現するため）である[†3]。FJ ではラムダ計算と同じように、仮引数がメソッド本体の中に複数回現れる場合、引数の値が複製されることになるかもしれない。しかし副作用がないので標準の Java 意味論との違いは観測できない。

以下はキャストの規則（E-CastNew）の動作例である。

```
(Pair)new Pair(new A(), new B())  ⟶  new Pair(new A(), new B())
```

いったんキャストの対象がオブジェクトに簡約されたら、コンストラクタのクラスがキャスト先のサブクラスであるか確認するのは簡単である。もし（今回のように）サブクラスであれば、簡約でキャストが取り除かれる。もし項 (A)new B() のようにサブクラスでないときは、この規則は適用されず、計算は**行き詰まり状態**となる。これは実行時エラーを表す。

計算を行き詰まり状態にしてしまう方法は三つある。クラスに宣言されていないフィールドを参照しようとすること、クラスに宣言されていないメソッドを呼び出そうとすること（「メッセージが理解されない」）、または、オブジェクトの実行時のクラスのスーパークラス以外の何かにキャストしようとすること、である。正しく型付けされたプログラムでは、最初の二つが起きないことを証明できる。また、プログラムがダウンキャスト（と「愚かなキャスト」、この技術的詳細については後で説明する）を含まず、かつ正しく型付けされているときは三つ目も起きないことを証明できる。

我々は標準的な値呼び評価戦略を採用する。上の二つ目の例の項を評価する各ステップを以下に示す。各ステップで、次に簡約される部分項に下線を引いている。

```
   ((Pair) (new Pair(new Pair(new A(),new B()), new A())
                .fst).snd
⟶  ((Pair)new Pair(new A(),new B())).snd
⟶  new Pair(new A(), new B()).snd
⟶  new B()
```

19.3 名前的型システムと構造的型システム

FJ の形式的な定義に進む前に、FJ（や Java）と、本書の主な焦点である型付きラムダ計算との基礎的な様式の違いについて調べておくべきである。この違いとは**型名**の立ち位置に関することである。

本章以前では、例を読みやすくするため、長くて複合的な型や複雑で複合的な型に短い名前をしばしば定義した。

```
NatPair = {fst:Nat, snd:Nat};
```

そのような定義は純粋に表面的なものである。名前 NatPair は {fst:Nat,snd:Nat} の単なる省略形であり、この二つはすべての文脈で相互に置き換え可能である。計算体系の形式化の中では、省略形を扱うことはなかった。

一方、Java では、広く使われているプログラミング言語の多くと同様、型定義がずっと重要な役割を果たす。Java プログラムで用いられる複合的な型はすべて名前を持つ。ローカル変数、フィールド、メソッドの仮引数の型を宣言するときは、必ず名前を与える。そういうところで、{fst:Nat,snd:Nat} のような「裸」の型を書くことはできない。

[†3] Abadi と Cardelli のオブジェクト計算 [4] に慣れた読者は、ς 簡約の規則によく似ているとわかるだろう。

これらの型名は Java の部分型関係において極めて重大な役割を果たす。(クラスやインターフェイスの定義で) 新しい名前が導入されるときは、導入されるものがどのクラスやインターフェイスを extends するかを、プログラマが明示的に宣言する (新しいクラスを既存のインターフェイスから作る場合は extends ではなく "implements")。コンパイラはこれらの宣言を検査して、導入されるクラスやインターフェイスが提供する機能が、確かにスーパークラス・スーパーインターフェイスの機能の拡張になっていることを保証する。この検査は型付きラムダ計算のレコード部分型付けに相当する。部分型関係は、宣言された直接の部分型関係の反射的推移的閉包として型**名**の間に定義される。もしある名前が他の名前の部分型になると宣言されていないなら、それは部分型ではない。

Java のように、名前が重要で、部分型関係が明示的に宣言される型システムを**名前的**という。本書で扱うほとんどの型システムのように、名前が本質的ではなく、部分型関係が型の構造の上に直接定義される型システムを**構造的**という。

名前的型システムは構造的型システムに比べて、利点も欠点もある。おそらく最も重要な利点は、名前的型システムの型名が型検査時だけでなく実行時にも有用であることである。名前的な言語の多くは、実行時の各オブジェクトを、そのオブジェクトの型名を持つヘッダワードでタグ付けする。このワードは具体的には、型を説明すると共に直接の上位型へのポインタを与える、実行時のデータ構造へのポインタで表現される。この型タグはさまざまな目的に用いることができる。実行時型検査 (例えば、Java の `instanceOf` 検査やダウンキャスト操作)、表示、データ構造をファイルに保存したりネットワーク越しに送信したりするためにバイナリ形式にマーシャリングすること、与えられたオブジェクトが提供するフィールドやメソッドをプログラムが動的に調査できるようにするリフレクション機能、などである。実行時の型タグは構造的型システムでも利用可能である (Glew [180]; League, Shao, and Trifonov [274]; League, Trifonov, and Shao [275] および、それらの参考文献を見よ) が、通常の型とは別の付加的機構をなす。名前的型システムでは、実行時の型タグはコンパイル時の型と同一視される。

名前的型システムの、あまり本質的ではないが嬉しい性質として、その定義の中で自分自身に言及するような型、**再帰型**を自然で直観的に説明できるということがある (再帰型については第 20 章で詳しく議論する)。このような型はリストや木などの一般的な構造を記述するために必要になるので、本格的なプログラミングで頻出する。名前的型システムは考えられる中で最も単純な方法で、このような型を扱える。`List` の定義の中で `List` を参照するのは他の型を参照するのと同じで、難しいことはない。実際、**相互に**再帰的な型ですら単純に書ける。型名の集合は最初から与えられているものとみなす。こうすることで、型 A の定義が名前 B を含み、B の定義が A を参照する場合に、「どちらが先に定義されたか」という問題が起きないようにしている。もちろん、構造的型システムでも再帰型は扱える。実際 ML のような、構造的型システムを持つ高水準言語では、通常、再帰型が他の機能と「セット」にされる。こうすることで再帰型は、プログラマにとっては、名前的な言語と全く同様に自然で簡単なものとなる。しかし型安全性の証明のような、より基礎研究的な目的を意図した計算体系では、再帰型を厳密に扱うために必要な仕組みはかなりの重荷となる。特に相互再帰型が許される場合に顕著である。名前的型システムだと再帰型が基本的に代償なく手に入る。このことは決定的な利点である。

名前的型システムには他にも、ある型が他の型の部分型かどうかを検査することがほぼ自明になるという利点がある。もちろんコンパイラは、宣言された部分型関係が安全かどうかを検証しなければならない。これは本質的には構造的な部分型関係と同じである。しかしこの検証は、型が定義されるごとに一回やるだけでよく、すべての部分型検査の最中にはやる必要はない。ただし、より本格的なコンパイラでは、この名前的な方式と構造的な方式の違いが性能に大きな影響を持つかどうか明らかではない。構造的型システムでもきちんと設計された型検査器なら、ほとんどの部分型

検査を単一の比較で済ませる表現技法を採用しているためである。174ページを見よ。

最後に、明示的な部分型宣言のよく挙げられる利点は、「偽の包摂関係」を防止することである。偽の包摂関係とは、ある型の値を、その型とは完全に別だが構造的には互換である型が期待されている場所に用いるようなプログラムを型検査器が拒絶し損ねるという問題である。この点に関しては、他の点より異論が多い。というのも、偽の包摂関係を防止するには（おそらくより良い）他の方法があるからである。例えば、単一コンストラクタデータ型（105ページ）や抽象データ型（第24章）である。

以上の利点（中でも、型タグの幅広い利便性や再帰型の扱いが簡単なこと）を考えると、名前的型システムが主流のプログラミング言語で標準となっていることは驚くにはあたらない。しかし、プログラミング言語の研究論文のほとんどは、構造的型システムに関心がある。

そうなっている直接の理由は、少なくとも再帰型がなければ、構造的型システムのほうが少し整然としていて、洗練されているからである。構造的なほうでは、型式は閉じた、独立したものとなる。つまり、型式の意味を理解するために必要な情報をすべて備えている。名前的型システムでは、型名とその定義についての大域的な情報を常に扱わないとならない。そのせいで、定義も証明もより冗長となってしまいがちである。

より興味深い理由は、研究論文が目を向けがちなのが、より発展的な機能（特に、パラメータ多相、抽象データ型、ユーザ定義の型演算子、ファンクタなど、型抽象に関する強力な仕組み）や、そういった機能を備えるMLやHaskellなどの言語であるからである。残念ながら、そういう機能は名前的型システムとあまり相性が良くない。例えば、`List(T)` のような型は、どう見ても複合的なように思われる。プログラムのどこかにコンストラクタ `List` の唯一の定義があり、`List(T)` の挙動を見るにはその定義を参照する必要がある。よって、`List(T)` を原子的な名前として扱うことはできない。名前的な言語には、このような「総称的」機能（Myers, Bank, and Liskov [332]; Agesen, Freund, and Mitchell [15]; Bracha, Odersky, Stoutamire, and Wadler [54]; Cartwright and Steele [91]; Stroustrup [447]） の拡張を持つものがある。しかしそういう拡張を施したものは、もはや純粋な名前的型システムではなく、二つのアプローチの少々複雑なハイブリッドである。発展的な型機能を備える言語の設計者は、構造的なアプローチを選ぶ傾向がある。

名前的型システムと構造的型システムの関係を完全に説明することは、現在進行中の研究課題として残っている。

19.4 定義

では、FJの形式的定義に移ろう。

文法

FJの文法を図19-1に示す。メタ変数A、B、C、D、Eは、クラス名の上を動く。`f` と `g` はフィールド名の上を動く。`m` はメソッド名の上を動く。`x` は仮引数名の上を動く。`s` と `t` は項の上を動く。`u` と `v` は値の上を動く。CLはクラス定義の上を動く。Kはコンストラクタ宣言の上を動く。Mはメソッド宣言の上を動く。特殊変数 `this` は変数の集合に含まれるが、メソッドの引数の名前として用いられることはないものとする。その代わり、すべてのメソッド宣言で暗に束縛されているものと考える。メソッド呼び出しの評価規則（図19-3の規則E-InvkNew）は、仮引数に実引数の値を代入するのに加え、`this` に適切なオブジェクトを代入する。

$\overline{\mathtt{f}}$ は $\mathtt{f}_1,\ldots,\mathtt{f}_n$ の省略記法とする（$\overline{\mathtt{C}}$, $\overline{\mathtt{x}}$, $\overline{\mathtt{t}}$ なども同様）。$\overline{\mathtt{M}}$ は（コンマなしの）$\mathtt{M}_1\ldots\mathtt{M}_n$ の省略記法とする。列の組に対する操作も同様に略記する。"$\overline{\mathtt{C}}\ \overline{\mathtt{f}}$" は "$\mathtt{C}_1\ \mathtt{f}_1,\ldots,\mathtt{C}_n\ \mathtt{f}_n$" の略記であ

構文		
CL ::=		クラス宣言：
	class C extends C {$\overline{\mathtt{C}}\ \overline{\mathtt{f}}$; K $\overline{\mathtt{M}}$}	
K ::=		コンストラクタ宣言：
	C($\overline{\mathtt{C}}\ \overline{\mathtt{f}}$) {super($\overline{\mathtt{f}}$); this.$\overline{\mathtt{f}}$=$\overline{\mathtt{f}}$;}	
M ::=		メソッド宣言：
	C m($\overline{\mathtt{C}}\ \overline{\mathtt{x}}$) {return t;}	
t ::=		項：
	x	変数
	t.f	フィールドアクセス
	t.m($\overline{\mathtt{t}}$)	メソッド呼び出し
	new C($\overline{\mathtt{t}}$)	オブジェクト生成
	(C) t	キャスト
v ::=		値：
	new C($\overline{\mathtt{v}}$)	オブジェクト生成

部分型付け **C<:D**

$$\mathtt{C} <: \mathtt{C}$$

$$\frac{\mathtt{C} <: \mathtt{D} \qquad \mathtt{D} <: \mathtt{E}}{\mathtt{C} <: \mathtt{E}}$$

$$\frac{CT(\mathtt{C}) = \mathtt{class\ C\ extends\ D\ \{...\}}}{\mathtt{C} <: \mathtt{D}}$$

▲図 19-1. Featherweight Java（構文と部分型付け）

り（ただし n は $\overline{\mathtt{C}}$ と $\overline{\mathtt{f}}$ の長さ）、"$\overline{\mathtt{C}}\ \overline{\mathtt{f}}$;" は "$\mathtt{C}_1\ \mathtt{f}_1$;...$\mathtt{C}_n\ \mathtt{f}_n$;" の略記であり、"this.$\overline{\mathtt{f}}$=$\overline{\mathtt{f}}$;" は "this.$\mathtt{f}_1$=$\mathtt{f}_1$;...this.$\mathtt{f}_n$=$\mathtt{f}_n$;" の略記である。フィールド宣言、仮引数名、メソッド宣言の列は、重複する名前を含まないものと仮定する。

宣言 class C extends D {$\overline{\mathtt{C}}\ \overline{\mathtt{f}}$; K $\overline{\mathtt{M}}$} は、D をスーパークラスに持つ、C という名前のクラスを導入する。この新しいクラスは、型 $\overline{\mathtt{C}}$ であるフィールド $\overline{\mathtt{f}}$ と、一つのコンストラクタ K と、一連のメソッド $\overline{\mathtt{M}}$ を持つ。C で宣言されるインスタンス変数は D やそのスーパークラスで宣言されたインスタンス変数に追加される。C で宣言されるインスタンス変数は、それらとは異なる名前でなければいけない[†4]。一方 C のメソッドは、D に既に存在する同名のメソッドをオーバーライドすることも、C に固有の新しい機能を追加することもできる。

コンストラクタ宣言 C($\overline{\mathtt{D}}\ \overline{\mathtt{g}}$, $\overline{\mathtt{C}}\ \overline{\mathtt{f}}$) {super($\overline{\mathtt{g}}$); this.$\overline{\mathtt{f}}$=$\overline{\mathtt{f}}$;} は、C のインスタンスのフィールドを初期化する方法を示している。その形式は C のインスタンス変数宣言とスーパークラスから完全に決められている。引数はインスタンス変数とちょうど同じ数だけ受け取ら**なくてはならない**。また、コンストラクタの本体は、引数 $\overline{\mathtt{g}}$ からフィールドを初期化するためスーパークラスのコンストラクタの呼び出しを含ま**なくてはならず**、その後で、C で宣言された同じ名前の新しいフィールドに引数 $\overline{\mathtt{f}}$ を代入し**なくてはならない**（これらの制約は図 19-4 のクラスの型付け規則で実現されている）。完全な Java では、サブクラスのコンストラクタはスーパークラスのコンストラクタの呼び出しを含まなければならない（スーパークラスのコンストラクタが引数を取る場合）。スーパークラス

[†4] Java では、スーパークラスのインスタンス変数は再宣言されるかもしれない。再宣言すると、元々あった変数は現在のクラスとそのサブクラスから隠される。FJ ではこの機能は省略されている。

のインスタンス変数を初期化するためにコンストラクタ本体から super を呼ぶのはそういう理由である。FJ を Java の文字通りのサブセットにすることにこだわらなかったら、super の呼び出しを削って、各コンストラクタがすべてのインスタンス引数を直接初期化するようにすることもできた。

メソッド宣言 D m($\overline{\mathtt{C}}$ $\overline{\mathtt{x}}$) {return t;} は、結果の型が D で、型 $\overline{\mathtt{C}}$ の仮引数 $\overline{\mathtt{x}}$ を持つ、m という名前のメソッドを導入する。メソッド本体は return t というただ一つの式である。t の中では変数 $\overline{\mathtt{x}}$ が束縛されている。特殊変数 this も t の中で束縛されているとみなす。

クラス表 *CT* は、クラス名 C からクラス宣言 CL への写像である。プログラムは、クラス表と項の組 (*CT*, t) である。以下では、記法を楽にするため、ある**決まった**クラス表 *CT* があるものと常に仮定する。

すべてのクラスは extends で宣言されたスーパークラスを持つ。ここで一つの疑問が生じる。Object クラスのスーパークラスは何だろうか。この問題を扱うには幾つか方法がある。最も簡単な方法は、Object を、クラス表の中に定義が現れない特別なクラス名とする方法である（今回もこれを採用する)。クラス表からフィールドを検索する補助関数は Object を特別扱いして、フィールドの空列 • を返すようになっている。Object はメソッドも持たないものと仮定する[†5]。

クラス表を見ると、クラス間の部分型関係を読み取れる。つまり部分型関係は、*CT* の中の extends 節で与えられた直接のサブクラス関係の反射的推移的閉包である。C が D の部分型であるとき、C <: D と書く。これは図 19-1 で形式的に定義される。

与えられたクラス表は以下の健全性条件を満たしているものと仮定する。(1) すべてのクラス C $\in$ *dom*(*CT*) に対して *CT*(C) = class C...、(2) Object $\notin$ *dom*(*CT*)、(3) *CT* のどこかに現れるすべてのクラス名 C（ただし Object 以外）に対して、C $\in$ *dom*(*CT*) であること、(4) *CT* によって作られる部分型関係の中に循環がないこと、つまり、<: 関係が反対称的であること。

クラス表で定義される型は、再帰的であることが**許されている**ことに注意されたい。これはクラス A の定義の中で、メソッドやインスタンス変数の型として A の名前を用いてもよいという意味である。クラス定義間の相互再帰も許されている。

演習 19.4.1. [★]　部分型付けのあるラムダ計算の S-Top 規則と同じように、Object がすべてのクラスの上位型であることを述べた規則があることを期待した読者がいるかもしれない。なぜここではそうする必要がないのだろうか。 □

補助的な定義

型付け規則と評価規則のために、補助的な定義を幾つか行う必要がある。これらは図 19-2 にまとめた。クラス C のフィールドの列を *fields*(C) と書く。これは、クラス C とそのスーパークラスで宣言されているフィールドすべてに対して、フィールドのクラスとフィールド名を組にした列 $\overline{\mathtt{C}}$ $\overline{\mathtt{f}}$ のことである。クラス C のメソッド m の型を、*mtype*(m, C) と書く。これは、引数の型の列 $\overline{\mathtt{B}}$ と結果の型 B との組 $\overline{\mathtt{B}}\to\mathtt{B}$ のことである。同様に、クラス C のメソッド m の本体を *mbody*(m, C) と書く。これは、仮引数の列 $\overline{\mathtt{x}}$ と項 t との組 ($\overline{\mathtt{x}}$, t) のことである。述語 *override*(m, D, $\overline{\mathtt{C}}\to\mathtt{C}_0$) は、引数の型が $\overline{\mathtt{C}}$ で結果の型が $\mathtt{C}_0$ であるメソッド m を D のサブクラスに定義可能かどうかを判断する。スーパークラスで宣言されている同名のメソッドをオーバーライドする場合は同じ型を持たなければならない。

[†5] 完全な Java では、クラス Object は実際にはメソッドを幾つか持っている。FJ ではこれらを無視する。

フィールドの探索 $\boxed{\mathit{fields}(\texttt{C}) = \overline{\texttt{C}}\ \overline{\texttt{f}}}$

$$\mathit{fields}(\texttt{Object}) = \bullet$$

$$\frac{\begin{array}{c} \mathit{CT}(\texttt{C}) = \texttt{class C extends D }\{\overline{\texttt{C}}\ \overline{\texttt{f}};\ \texttt{K}\ \overline{\texttt{M}}\} \\ \mathit{fields}(\texttt{D}) = \overline{\texttt{D}}\ \overline{\texttt{g}} \end{array}}{\mathit{fields}(\texttt{C}) = \overline{\texttt{D}}\ \overline{\texttt{g}}, \overline{\texttt{C}}\ \overline{\texttt{f}}}$$

メソッドの型の探索 $\boxed{\mathit{mtype}(\texttt{m}, \texttt{C}) = \overline{\texttt{C}} \rightarrow \texttt{C}}$

$$\frac{\begin{array}{c} \mathit{CT}(\texttt{C}) = \texttt{class C extends D }\{\overline{\texttt{C}}\ \overline{\texttt{f}};\ \texttt{K}\ \overline{\texttt{M}}\} \\ \texttt{B m }(\overline{\texttt{B}}\ \overline{\texttt{x}})\ \{\texttt{return t;}\} \in \overline{\texttt{M}} \end{array}}{\mathit{mtype}(\texttt{m}, \texttt{C}) = \overline{\texttt{B}} \rightarrow \texttt{B}}$$

$$\frac{\begin{array}{c} \mathit{CT}(\texttt{C}) = \texttt{class C extends D }\{\overline{\texttt{C}}\ \overline{\texttt{f}};\ \texttt{K}\ \overline{\texttt{M}}\} \\ \texttt{m}\ \text{は}\ \overline{\texttt{M}}\ \text{の中で定義されていない} \end{array}}{\mathit{mtype}(\texttt{m}, \texttt{C}) = \mathit{mtype}(\texttt{m}, \texttt{D})}$$

メソッドの本体の探索 $\boxed{\mathit{mbody}(\texttt{m}, \texttt{C}) = (\overline{\texttt{x}}, \texttt{t})}$

$$\frac{\begin{array}{c} \mathit{CT}(\texttt{C}) = \texttt{class C extends D }\{\overline{\texttt{C}}\ \overline{\texttt{f}};\ \texttt{K}\ \overline{\texttt{M}}\} \\ \texttt{B m }(\overline{\texttt{B}}\ \overline{\texttt{x}})\ \{\texttt{return t;}\} \in \overline{\texttt{M}} \end{array}}{\mathit{mbody}(\texttt{m}, \texttt{C}) = (\overline{\texttt{x}}, \texttt{t})}$$

$$\frac{\begin{array}{c} \mathit{CT}(\texttt{C}) = \texttt{class C extends D }\{\overline{\texttt{C}}\ \overline{\texttt{f}};\ \texttt{K}\ \overline{\texttt{M}}\} \\ \texttt{m}\ \text{は}\ \overline{\texttt{M}}\ \text{の中で定義されていない} \end{array}}{\mathit{mbody}(\texttt{m}, \texttt{C}) = \mathit{mbody}(\texttt{m}, \texttt{D})}$$

メソッドの正当なオーバーライド $\boxed{\mathit{override}(\texttt{m}, \texttt{D}, \overline{\texttt{C}} \rightarrow \texttt{C}_0)}$

$$\frac{\mathit{mtype}(\texttt{m}, \texttt{D}) = \overline{\texttt{D}} \rightarrow \texttt{D}_0\ \text{ならば、}\ \overline{\texttt{C}} = \overline{\texttt{D}}\ \text{かつ}\ \texttt{C}_0 = \texttt{D}_0}{\mathit{override}(\texttt{m}, \texttt{D}, \overline{\texttt{C}} \rightarrow \texttt{C}_0)}$$

▲ 図 19-2. Featherweight Java（補助的な定義）

評価 $\boxed{\texttt{t} \longrightarrow \texttt{t}'}$

$$\frac{\mathit{fields}(\texttt{C}) = \overline{\texttt{C}}\ \overline{\texttt{f}}}{(\texttt{new C}(\overline{\texttt{v}}))\texttt{.f}_i \longrightarrow \texttt{v}_i} \quad \text{(E-ProjNew)}$$

$$\frac{\mathit{mbody}(\texttt{m}, \texttt{C}) = (\overline{\texttt{x}}, \texttt{t}_0)}{\begin{array}{c}(\texttt{new C}(\overline{\texttt{v}}))\texttt{.m}(\overline{\texttt{u}}) \\ \longrightarrow [\overline{\texttt{x}} \mapsto \overline{\texttt{u}}, \texttt{this} \mapsto \texttt{new C}(\overline{\texttt{v}})]\texttt{t}_0\end{array}} \quad \text{(E-InvkNew)}$$

$$\frac{\texttt{C <: D}}{(\texttt{D})(\texttt{new C}(\overline{\texttt{v}})) \longrightarrow \texttt{new C}(\overline{\texttt{v}})} \quad \text{(E-CastNew)}$$

$$\frac{\texttt{t}_0 \longrightarrow \texttt{t}_0'}{\texttt{t}_0\texttt{.f} \longrightarrow \texttt{t}_0'\texttt{.f}} \quad \text{(E-Field)}$$

$$\frac{\texttt{t}_0 \longrightarrow \texttt{t}_0'}{\texttt{t}_0\texttt{.m}(\overline{\texttt{t}}) \longrightarrow \texttt{t}_0'\texttt{.m}(\overline{\texttt{t}})} \quad \text{(E-Invk-Recv)}$$

$$\frac{\texttt{t}_i \longrightarrow \texttt{t}_i'}{\begin{array}{c}\texttt{v}_0\texttt{.m}(\overline{\texttt{v}},\ \texttt{t}_i,\ \overline{\texttt{t}}) \\ \longrightarrow \texttt{v}_0\texttt{.m}(\overline{\texttt{v}},\ \texttt{t}_i',\ \overline{\texttt{t}})\end{array}} \quad \text{(E-Invk-Arg)}$$

$$\frac{\texttt{t}_i \longrightarrow \texttt{t}_i'}{\begin{array}{c}\texttt{new C}(\overline{\texttt{v}},\ \texttt{t}_i,\ \overline{\texttt{t}}) \\ \longrightarrow \texttt{new C}(\overline{\texttt{v}},\ \texttt{t}_i',\ \overline{\texttt{t}})\end{array}} \quad \text{(E-New-Arg)}$$

$$\frac{\texttt{t}_0 \longrightarrow \texttt{t}_0'}{(\texttt{C})\texttt{t}_0 \longrightarrow (\texttt{C})\texttt{t}_0'} \quad \text{(E-Cast)}$$

▲ 図 19-3. Featherweight Java（評価）

評価

標準的な値呼び操作的意味論を用いる（図 19-3）。三つの規則（フィールド参照、メソッド呼び出し、キャスト）は 19.2 節で説明した。残りの規則は値呼び戦略を形式化している。評価器の正常終了の結果となりうる値は、$\texttt{new C}(\overline{\texttt{v}})$ という形の、完全に評価されたオブジェクト生成の項である。

キャスト操作の実行時の挙動は、キャストしようとしているオブジェクトの実際の型がキャストの中で宣言された型の部分型かどうかをテストすることである。もし部分型ならば、キャスト操作は捨てられ、そのオブジェクト自身が結果となる。これは Java の意味論に完全に対応する。実行時

項の型付け　$\boxed{\Gamma \vdash \mathtt{t} : \mathtt{C}}$

$$\frac{\mathtt{x}{:}\mathtt{C} \in \Gamma}{\Gamma \vdash \mathtt{x} : \mathtt{C}} \quad \text{(T-VAR)}$$

$$\frac{\Gamma \vdash \mathtt{t}_0 : \mathtt{C}_0 \qquad \mathit{fields}(\mathtt{C}_0) = \overline{\mathtt{C}}\ \overline{\mathtt{f}}}{\Gamma \vdash \mathtt{t}_0.\mathtt{f}_i : \mathtt{C}_i} \quad \text{(T-FIELD)}$$

$$\frac{\begin{array}{c}\Gamma \vdash \mathtt{t}_0 : \mathtt{C}_0 \\ \mathit{mtype}(\mathtt{m}, \mathtt{C}_0) = \overline{\mathtt{D}} \to \mathtt{C} \\ \Gamma \vdash \overline{\mathtt{t}} : \overline{\mathtt{C}} \qquad \overline{\mathtt{C}} <: \overline{\mathtt{D}}\end{array}}{\Gamma \vdash \mathtt{t}_0.\mathtt{m}(\overline{\mathtt{t}}) : \mathtt{C}} \quad \text{(T-INVK)}$$

$$\frac{\begin{array}{c}\mathit{fields}(\mathtt{C}) = \overline{\mathtt{D}}\ \overline{\mathtt{f}} \\ \Gamma \vdash \overline{\mathtt{t}} : \overline{\mathtt{C}} \qquad \overline{\mathtt{C}} <: \overline{\mathtt{D}}\end{array}}{\Gamma \vdash \mathtt{new}\ \mathtt{C}(\overline{\mathtt{t}}) : \mathtt{C}} \quad \text{(T-NEW)}$$

$$\frac{\Gamma \vdash \mathtt{t}_0 : \mathtt{D} \qquad \mathtt{D} <: \mathtt{C}}{\Gamma \vdash (\mathtt{C})\mathtt{t}_0 : \mathtt{C}} \quad \text{(T-UCAST)}$$

$$\frac{\Gamma \vdash \mathtt{t}_0 : \mathtt{D} \qquad \mathtt{C} <: \mathtt{D} \qquad \mathtt{C} \neq \mathtt{D}}{\Gamma \vdash (\mathtt{C})\mathtt{t}_0 : \mathtt{C}} \quad \text{(T-DCAST)}$$

$$\frac{\begin{array}{c}\Gamma \vdash \mathtt{t}_0 : \mathtt{D} \qquad \mathtt{C} \not<: \mathtt{D} \qquad \mathtt{D} \not<: \mathtt{C} \\ \text{愚かさの警告}\end{array}}{\Gamma \vdash (\mathtt{C})\mathtt{t}_0 : \mathtt{C}} \quad \text{(T-SCAST)}$$

メソッドの型付け　$\boxed{\mathtt{M}\ \mathtt{OK}\ \mathtt{in}\ \mathtt{C}}$

$$\frac{\begin{array}{c}\overline{\mathtt{x}} : \overline{\mathtt{C}}, \mathtt{this} : \mathtt{C} \vdash \mathtt{t}_0 : \mathtt{E}_0 \qquad \mathtt{E}_0 <: \mathtt{C}_0 \\ \mathit{CT}(\mathtt{C}) = \mathtt{class}\ \mathtt{C}\ \mathtt{extends}\ \mathtt{D}\ \{\ldots\} \\ \mathit{override}(\mathtt{m}, \mathtt{D}, \overline{\mathtt{C}} \to \mathtt{C}_0)\end{array}}{\mathtt{C}_0\ \mathtt{m}\ (\overline{\mathtt{C}}\ \overline{\mathtt{x}})\ \{\mathtt{return}\ \mathtt{t}_0;\}\ \mathtt{OK}\ \mathtt{in}\ \mathtt{C}}$$

クラスの型付け　$\boxed{\mathtt{C}\ \mathtt{OK}}$

$$\frac{\begin{array}{c}\mathtt{K} = \mathtt{C}(\overline{\mathtt{D}}\ \overline{\mathtt{g}}, \overline{\mathtt{C}}\ \overline{\mathtt{f}}) \\ \{\mathtt{super}(\overline{\mathtt{g}});\ \mathtt{this}.\overline{\mathtt{f}} = \overline{\mathtt{f}};\} \\ \mathit{fields}(\mathtt{D}) = \overline{\mathtt{D}}\ \overline{\mathtt{g}} \qquad \overline{\mathtt{M}}\ \mathtt{OK}\ \mathtt{in}\ \mathtt{C}\end{array}}{\mathtt{class}\ \mathtt{C}\ \mathtt{extends}\ \mathtt{D}\ \{\overline{\mathtt{C}}\ \overline{\mathtt{f}};\ \mathtt{K}\ \overline{\mathtt{M}}\}\ \mathtt{OK}}$$

▲ 図 19-4.　Featherweight Java（型付け）

キャストはオブジェクトにいかなる変更も行わない。成功するか、さもなくば失敗して例外を投げるかの二択しかないためである。FJ ではキャストが失敗したら、例外を投げるのではなく、単に行き詰まり状態になる。適用できる評価規則がないためである。

型付け

項、メソッド宣言、クラス宣言の型付け規則を図 19-4 に示す。環境 Γ は、変数から型への有限の写像であり、$\overline{\mathtt{x}}{:}\overline{\mathtt{C}}$ と書く。項の型付け判断式は $\Gamma \vdash \mathtt{t} : \mathtt{C}$ という形で、「環境 Γ において、項 t は型 C を持つ」と読む。型付け規則は構文主導であり[†6]、項の形ごとに規則が一つある。ただし、キャストには三つの規則がある（後で議論する）。コンストラクタとメソッド呼び出しの型付け規則は、各実引数が対応する仮引数に注釈されている型の部分型を持っていることを検査する。列の型付け判断式に、自明な省略記法を用いる。$\Gamma \vdash \overline{\mathtt{t}} : \overline{\mathtt{C}}$ は $\Gamma \vdash \mathtt{t}_1 : \mathtt{C}_1, \ldots, \Gamma \vdash \mathtt{t}_n : \mathtt{C}_n$ の省略であり、$\overline{\mathtt{C}} <: \overline{\mathtt{D}}$ は $\mathtt{C}_1 <: \mathtt{D}_1, \ldots, \mathtt{C}_n <: \mathtt{D}_n$ の省略である。

FJ のちょっとした技術的革新は、「愚かな」キャストの導入である。キャストには三つの規則がある。**アップキャスト**では、キャストの対象がキャスト先のサブクラスである。**ダウンキャスト**では、キャスト先がキャストの対象のサブクラスである。**愚かな**キャストでは、キャスト先とキャストの対象が無関係である。Java コンパイラは愚かなキャストを含む項を、正しく型付けされないものとして拒絶する。しかし FJ では、小ステップ意味論の型保存定理として型安全性を定式化する以上、愚かなキャストを認めなければならない。これは、愚かなキャストを含まない項が愚かなキャストを含む項に簡約されることがあるからである。例えば、19.2 節で定義したクラス A と B 用いて、

[†6] 我々は完全な Java にならって、型付け関係のアルゴリズム的な形式化を選んだ。興味深いことに、完全な Java ではこの選択は避けて通れない。演習 19.4.6 を見よ。

(A)(Object)new B() ⟶ (A)new B()

という例を考えるとよい。我々は、愚かなキャストの型規則（T-SCast）に前提（愚かさの警告）を含めることで、愚かなキャストの特殊な本質を示している。FJの型付けが正しいJavaの型付けに対応するのは、この規則を含まないときだけである。

メソッド宣言の型付け判断式は、M OK in Cという形式を持ち、「メソッド宣言Mは、それがクラスCの中にあるなら正しい形式である」と読む。メソッド本体で項の型付け関係を用いている。項の中に現れうる自由変数は、メソッドの仮引数（その型は宣言されたもの）と、特殊変数this（その型はC）である。

クラス宣言の型付け判断式はCL OKという形式を持ち、「クラス宣言CLは正しい形式をしている」と読む。コンストラクタがスーパークラスのフィールドにsuperを適用し、そのクラスで宣言されているフィールドを初期化していること、そして、そのクラスの各メソッド宣言が問題ないことを検査している。

項の型は、それが呼び出す任意のメソッドの型に依存するかもしれない。メソッドの型は、（その本体である）項の型に依存するかもしれない。よって、不適切な循環的定義になっていないことを検査すべきである。実際のところ、そういう心配はない。各メソッドの型は明示的に宣言されているので、循環はしない。クラス表中のクラスすべてを検査するより先にクラス表を読み込んで、それを型検査に使うことができる。最終的に各クラスが検査されればそれでよい。

演習 19.4.2. [★★]　FJの設計に関するたくさんの決定においては、FJをJavaのサブセットにしたいということを念頭に置いた。つまり、正しく型付けされたFJプログラムは正しく型付けされたJavaプログラムに、そうでないFJプログラムはそうでないJavaプログラムになり、正しく型付けされたプログラムは同じように振る舞うということである。仮にこの要求を捨てたとしよう。つまり、欲しかったのは単に**Java風の**核となる計算体系だったとする。あなたなら、FJをより簡潔に、もしくはより洗練されたものにするために、設計をどのように変えるだろうか。 □

演習 19.4.3. [推奨, ★★★ ↛]　簡単のため、新しい値をオブジェクトのフィールドに破壊的に代入する操作はFJから省略された。しかし代入は、計算体系の基本的な特色を大きく変えることなく追加できる。第13章の参照の扱い方を参考にして、これをやってみよ。 □

演習 19.4.4. [★★★ ↛]　第14章の例外の扱い方を参考にして、Javaのthrowやtryのようなもので FJを拡張せよ。 □

演習 19.4.5. [★★ ↛]　完全なJavaと同じように、FJでは型付け関係をアルゴリズム的な形式で表現している。包摂規則はない。その代わり、他の幾つかの規則が部分型検査を前提として含んでいる。これらの前提の多く、またはすべてを消し、単一の包摂規則を導入して、型システムをより宣言的に形式化し直すことはできるだろうか。 □

演習 19.4.6. [★★★]　完全なJavaは、クラスと**インターフェイス**の両方を提供する。インターフェイスはメソッドの型を指定するが、実装は持たない。インターフェイスが便利なのは、強力な、木構造でない部分型関係を提供することである。各クラスはスーパークラスを一つしか持たない（そこからインスタンス変数とメソッド本体を継承する）が、インターフェイスは任意の数だけ付加的に実装できる。

Javaにはインターフェイスが存在するので、型付け可能な項に対して一意の（最小の）型を与えてくれる、型付け関係のアルゴリズム的な形式化を選ばなければならない。その理由は、条件式

（Java では $\mathtt{t_1\ ?\ t_2\ :\ t_3}$ と書かれる）とインターフェイスの間の相互作用である。

(1) Java 流のインターフェイスで FJ を拡張する方法を示せ。
(2) インターフェイスが存在するなら、部分型関係が結びの下で必ずしも閉じているとは限らないことを示せ。（171 ページを思い出そう。結びの存在は、条件式の最小型付け性に重要な役割を果たしていた。）
(3) 条件式に対する Java の型付け規則はどのようなものだろうか。それは妥当なものだろうか。

□

演習 19.4.7. [★★★] FJ は Java の `this` キーワードを持つが、`super` は持たない。追加する方法を示せ。 □

19.5 性質

FJ では標準的な型保存定理が証明できる。

定理 19.5.1. [**保存**] $\Gamma \vdash \mathtt{t} : \mathtt{C}$ かつ $\mathtt{t} \longrightarrow \mathtt{t}'$ ならば、ある $\mathtt{C}' <: \mathtt{C}$ に対して $\Gamma \vdash \mathtt{t}' : \mathtt{C}'$ □

[証明] 演習[★★★]。 □

標準的な進行定理の変形も証明できる。もしプログラムが正しく型付けされていれば、それが行き詰まり状態になる唯一の方法はキャストできない状態に到達した場合だけである。**評価文脈**の仕組みを用いて、キャスト失敗を同定する。

補題 19.5.2. t が正しく型付けされた項だとする。

(1) $\mathtt{t} = \mathtt{new\ C_0(\overline{t}).f}$ ならば、$\mathit{fields}(\mathtt{C_0}) = \overline{\mathtt{C}}\ \overline{\mathtt{f}}$ かつ $\mathtt{f} \in \overline{\mathtt{f}}$
(2) $\mathtt{t} = \mathtt{new\ C_0(\overline{t}).m(\overline{s})}$ ならば、$\mathit{mbody}(\mathtt{m}, \mathtt{C_0}) = (\overline{\mathtt{x}}, \mathtt{t_0})$ かつ $|\overline{\mathtt{x}}| = |\overline{\mathtt{s}}|$ □

[証明] 単純。 □

定義 19.5.3. FJ の**評価文脈**の集合は以下のように定義される。

E ::=	評価文脈：
$[\,]$	穴
$E\mathtt{.f}$	フィールド参照
$E\mathtt{.m(\overline{t})}$	メソッド呼び出し（レシーバ）
$\mathtt{v.m(\overline{v},}E\mathtt{,\overline{t})}$	メソッド呼び出し（引数）
$\mathtt{new\ C(\overline{v},}E\mathtt{,\overline{t})}$	オブジェクト生成（引数）
$\mathtt{(C)}E$	キャスト

評価文脈は、[ ] で示される穴をどこかに含む項である。E の中の穴を t で置き換えて得られる普通の項を $E[\mathtt{t}]$ と書く。 □

評価文脈は「次に簡約される部分項」の概念を表している。$\mathtt{t} \longrightarrow \mathtt{t}'$ ならば、E-ProjNew、E-InvkNew、E-CastNew のいずれかの計算規則で $\mathtt{r} \longrightarrow \mathtt{r}'$ が成り立つような一意の E、r、r′ に対して、t と t′ を $\mathtt{t} = E[\mathtt{r}]$ と $\mathtt{t}' = E[\mathtt{r}']$ として表現できるということである。

定理 19.5.4. [**進行**] t が、閉じた、正しく型付けされた正規形であるとする。このとき (1) t は値

であるか、(2) ある評価文脈 E に対し、t を t = E[(C)(new D($\overline{\text{v}}$))]（ただし D $\not<:$ C）と表現できる。 □

[証明] 型付け導出に関する単純な帰納法。 □

進行の性質はもう少し強くできる。もし t がアップキャストしか含まないならば、行き詰まり状態になることはない（そして、元々のプログラムがアップキャストしか含まないならば、評価中にアップキャスト以外のキャストが現れることはない)。しかし一般的には、オブジェクトの静的な型を**低いほうへ**変えるキャストを使いたいので、キャストが実行時に失敗しうる可能性を受け入れなければならない。完全な Java ではもちろん、キャスト失敗でプログラム全体を止めることはせず、例外ハンドラで捕捉できる例外を生成する。

演習 19.5.5. [★★★ ↛] ラムダ計算の型検査器のどれかを出発点として、Featherweight Java の型検査器とインタプリタを構築せよ。 □

演習 19.5.6. [★★★★ ↛] 元々の FJ の論文（Igarashi, Pierce, and Wadler [231]）は、GJ 風の多相型も形式化している。演習 19.5.5 の型検査器とインタプリタを拡張して、これらの機能を持たせてみよ。（この演習に取り組む前に、第 23 章、第 25 章、第 26 章、第 28 章を読む必要があるだろう。） □

19.6 オブジェクト表現 対 組み込みのオブジェクト

簡単なオブジェクト指向言語の意味論と型付けに対する、対照的な二つのアプローチを見てきた。第 18 章では、レコード、参照、部分型付けの機能を備える単純型付きラムダ計算の機能を組み合わせて用いて、オブジェクト、クラス、継承を表現した。本章では、オブジェクトとクラスを組み込みの仕組みとして持つ単純な言語を直接的に説明した。

これらのアプローチにはどちらも使い道がある。オブジェクトの表現を研究すると、カプセル化や再利用の基礎的な仕組みがあらわになり、同じ目的の他の仕組みとの比較ができるようになる。また、より低水準な言語にオブジェクトを翻訳するコンパイル方法を考える手助けになったり、オブジェクトと他の言語機能の間の相互作用を理解する手助けになったりもする。一方、オブジェクトを組み込み機能として扱うと、操作的意味論や型付けの振る舞いを直接的に議論できる。この種の説明は、高水準言語の設計やドキュメント化に適した道具となる。

究極的にはもちろん、双方の視点を併せ持つことが望ましい。つまり、以下のような高水準言語があるとよい。まず、オブジェクトやクラスのための機能を組み込みで含み、固有の型付け規則と操作的意味論を持つこと。さらに、その言語から、レコードと関数しか持たない何らかの低水準言語（あるいは実際のところ、レジスタとポインタおよび命令列しか持たない、もっと低水準な言語）への翻訳が可能であること。そして、その翻訳が元の高水準言語の実装として正しいこと（すなわち、元の高水準言語の評価と型付けの性質を保存すること）が証明されていること。この課題はさまざまな形で取り組まれてきた。FJ 自身は League, Trifonov, and Shao [275] で、オブジェクト指向の核となる他の計算体系は Hofmann and Pierce [215] や、Bruce [59; 60]、Abadi, Cardelli, and Viswanathan [9] などで行われている。

19.7 注記

この章は、Igarashi, Pierce, and Wadler [231] による FJ の最初の論文を典拠としている。大きな違いは、本書では他の章との一貫性のため、値呼び操作的意味論を用いたことである。元々の論文では非決定的簡約関係を用いていた。

Java のサブセットに対する型安全性の証明は幾つかある。最も初期のもののは、Drossopoulou, Eisenbach, and Khurshid [140] で、後に Syme [451] によって機械的に検証された技法を用いて、逐次的 Java の相当量のサブセットに対して安全性を証明した。FJ 同様、小ステップ操作的意味論を用いているが、キャストを丸ごと省略することで、「愚かなキャスト」の微妙さを避けている。Nipkow and Oheimb [339] は、いくぶん大きな核言語に対する安全性の、機械的に検証された証明を与えた。彼らの言語はキャストを含むが、「大ステップ」操作的意味論を用いて形式化されていて、愚かなキャスト問題は避けられている。Flatt, Krishnamurthi, and Felleisen [158; 159] は小ステップ意味論を用いて破壊的代入とキャストのある言語を形式化している。FJ 同様、愚かなキャストを扱っている。彼らの体系は FJ よりやや大きく（文法、型付け、操作的意味論の規則はだいたい三倍くらいのスペースを要する）、それに対応するように安全性の証明も大きいが、複雑さは似たようなものである。

これら三つの研究の中では、Flatt, Krishnamurthi, and Felleisen [158; 159] が、ある重要な意味で FJ に最も近い。彼らの目標は我々と同様、核となる計算にはできる限り**小さい**ものを選び、特定の目的に応じた Java の機能を捉えることである。その目的とは彼らの場合、Common Lisp 流の mixin で Java を拡張したものを解析することである。一方、他の二つの体系の目標は、Java のできる限り**大きい**サブセットを含めることだった。というのも彼らの主な興味は、Java 自体の安全性を証明することだからである。

オブジェクト指向言語の基礎に関する文献には、クラスベースのオブジェクト指向言語の形式化に関する論文が多数ある。それらは、クラスを組み込みとするか（例えば Wand [489]、Bruce [59]、Bono, Patel, Shmatikov, and Mitchell [53]、Bono, Patel, and Shmatikov [52]）、クラスをより低水準な仕組みに翻訳している（例えば Fisher and Mitchell [154]、Bono and Fisher [51]、Abadi and Cardelli [4]、Pierce and Turner [373]）。

ある種の**メソッドオーバーライド**や**委譲**でクラスを置き換えたオブジェクト指向言語のモデルを考える関連研究もある（Ungar and Smith [471]）。個々のオブジェクトは、他の**オブジェクト**から挙動を継承する。扱う概念の集合が小さいので、結果的な計算体系はクラスベースの言語より多少簡単になる傾向がある。その中でも、最もよく研究され、広く知られているものは Abadi と Cardelli の**オブジェクト計算** [4] である。Fisher, Honsell, and Mitchell [152] が研究したものも有名である。

マルチメソッドとして知られる、オブジェクト、クラス、継承に対するかなり異なるアプローチがある。これは Castagna, Ghelli, and Longo [93] が形式化した。178 ページの脚注にさらなる参考文献がある。

> どんな大きな言語の中にも、そこから逃げ出そうとしている小さな言語がある。……
> —*Igarashi, Pierce, and Wadler, 1999* [231]

> どんな大きなプログラムの中にも、そこから逃げ出そうとしている小さなプログラムがある。……　—*Tony Hoare, Efficient Production of Large Programs, 1970*

> 私はでぶだが、中は痩せている。
> どんなでぶの男の中にも痩せた男がいる、と思ったことはないか？
> —*George Orwell, Coming Up For Air, 1939*

第Ⅳ部

再帰型

第20章

再帰型

本章で扱う体系は、再帰型を持つ単純型付きラムダ計算である。例では、これまでの章で導入したさまざまな機能を用いる。対応する検査器は fullequirec である。20.2 節に関しては、対応する検査器は fullisorec である。

11.12 節では、単純な型システムを拡張して、T 型の要素のリストからなる型構築子 List(T) を持たせた。リストは、キュー、二分木、ラベル付き木、抽象構文木などを含む多数のよくある構造の一例にすぎない。これらの構造はいくらでも大きなサイズになりうるが、単純で規則的な構造になっている。例えば、List(Nat) の要素は nil、もしくは、数と別の List(Nat) からなる二つ組（「cons セル」）のいずれかである。明らかに、それらの構造すべてを個別のプリミティブな言語機能として提供するのは合理的ではない。むしろ、必要に応じ、それらをより単純な構成要素から定義できる一般的な機構こそが必要である。この機構は**再帰型**と呼ばれる。

数のリストの型を再度考えてみよう[†1]。リストが nil もしくは二つ組であることは、11.10 節と 11.7 節で導入したバリアント型と組型を用いて表現できる。

```
NatList = <nil:Unit, cons:{...,...}>;
```

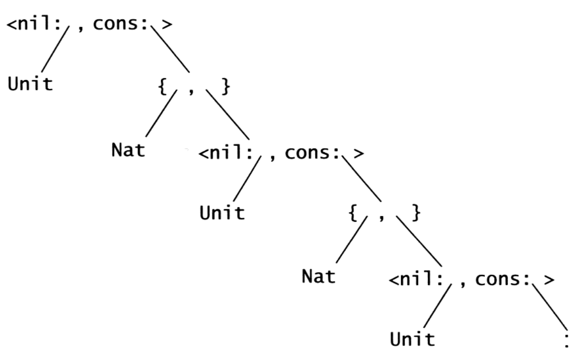
nil の保持するデータ値は自明なものである。これは nil タグそれ自体が既に空リストについて知るべきすべてを物語っているためである。一方、cons タグが保持する値は、数と別のリストの二つ組である。この二つ組の一つ目の要素は Nat 型を持つ。

```
NatList = <nil:Unit, cons: {Nat, ...}>;
```

二つ目の要素は数のリスト、すなわち今定義している NatList 型それ自体の要素である。

```
NatList = <nil:Unit, cons:{Nat,NatList}>;
```

この等式は普通の意味での定義ではない。すなわち、既に意味を理解している表現に対して新たな名前を与えているのではない。というのは、右辺が今定義しようとしている名前それ自体に言及しているためである。むしろこの等式は、無限木の仕様として考えられる。



無限木の型を定めるこの再帰的な等式は、40 ページの再帰的な factorial 関数を定める等式に似ている。その場合と同様に「ループ」を = の右辺へと移動することで、きちんとした定義にすると

†1 これ以降本章では、任意の型 T の要素からなるリストの単一かつ総称的な定義をする方法は扱わない。これを扱うには、第 29 章で導入する**型演算子**の機構も必要になる。

便利である[†2]。そのために、型のための明示的な再帰演算子 μ を導入する。

```
NatList = μX. <nil:Unit, cons:{Nat,X}>;
```

直観的には、この定義は「NatList を、等式 X = <nil:Unit, cons:{Nat,X}> を満たす無限型とする」と解釈する。

実際には 20.2 節で見るように、再帰型を形式化するには、**同値再帰**と**同型再帰**という、多少異なる二通りの方法がある。後者ではプログラマがより多くの型注釈を付けて型検査器を手助けしなくてはならない。次節のプログラミング例では、より軽量な同値再帰の形式を用いる。

20.1　例

リスト

まず最初に、上で始めた数のリストの例を片付けよう。リストを用いてプログラミングするには、定数 nil、既存のリストの先頭に新たな要素を加える構築子（constructor）である cons、リストを取りブール値を返す isnil 演算、非空のリストから先頭の要素（head）と残りの部分（tail）を取り出す分解子 hd と tl が必要となる。図 11-13 ではこれらすべてを組み込みの演算として定義していた。今度はこれらをより単純な部品から構成する。

nil と cons の定義は、2 フィールドのバリアント型としての NatList の定義から直接得られる。

```
  nil = <nil=unit> as NatList;
▸ nil : NatList

  cons = λn:Nat. λl:NatList. <cons={n,l}> as NatList;
▸ cons : Nat → NatList → NatList
```

(11.10 節で <l=t> as T という形の式でバリアント型の値を導入していたのを思い出そう。ここで値 t はラベル l でタグ付けされ、バリアント型 T へと「入射」されている。さらに、ここで型検査器は再帰型 NatList をバリアント型 <nil:Unit, cons:{Nat,NatList}> へと自動的に「展開」していることに注意されたい。)

リストの基本演算として残るは、リストの構造の確認や適切な部分の取り出しにかかわるものである。それらはすべて case を用いて実装できる。

```
  isnil = λl:NatList. case l of
                            <nil=u> ⇒ true
                          | <cons=p> ⇒ false;
▸ isnil : NatList → Bool

  hd = λl:NatList. case l of <nil=u> ⇒ 0 | <cons=p> ⇒ p.1;
▸ hd : NatList → Nat

  tl = λl:NatList. case l of <nil=u> ⇒ l | <cons=p> ⇒ p.2;
▸ tl : NatList → NatList
```

空リストに対する hd と tl の結果は、それぞれ 0 と空リストとして定義することに適当に決めてしまった。別の選択肢として、これらの場合には例外を発生させてもよいだろう。

これらすべての定義を組み合わせて用いることで、リスト要素の総和を計算する再帰関数を以下

[†2] この移動が便利なのは、それにより再帰型を名前を付けることなく扱えるようになるためである。しかし、明示的に名付けられた再帰型を用いることにも利点がある。これについては、19.3 節の**名前的**型システム対**構造的**型システムの議論を参照せよ。

のように記述できる。

```
sumlist = fix (λs:NatList→Nat. λl:NatList.
                  if isnil l then 0 else plus (hd l) (s (tl l)));
▸ sumlist : NatList → Nat

mylist = cons 2 (cons 3 (cons 5 nil));
sumlist mylist;
▸ 10 : Nat
```

ここで、NatList それ自体は無限に長い型式であるものの、その要素はすべて**有限**のリストであることに注意されたい（なぜならば、二つ組化とタグ付けのプリミティブ、あるいは値呼びの fix を用いて無限に大きい構造を作る方法はないためである）。

演習 20.1.1. [★★] ラベル付き二分木を表す一つの方法として、木を、（ラベルを持たない）葉、もしくは、数値のラベルと二つの子を持つ内部ノードの、いずれかとして定義する。型 NatTree と、木の構築・分解・検査を行う適切な演算を定義せよ。また、深さ優先走査を行い、見つかったラベルのリストを返す関数を記述せよ。記述したコードは検査器 fullequirec を用いてテストせよ。 □

空腹関数

他には、「空腹関数（hungry function）」は、再帰型のいくぶん巧妙な使い方を説明するのに良い例だろう。この関数は数の引数を幾つでも受け取ることができ、常に、さらなる引数を乞う新たな空腹関数を返す。

```
Hungry = μA. Nat→A;
```

この型の要素は fix 演算子を用いて定義できる。

```
f = fix (λf: Nat→Hungry. λn:Nat. f);
▸ f : Hungry

f 0 1 2 3 4 5;
▸ <fun> : Hungry
```

ストリーム

上の Hungry 型の変種でもっと役立つものとして、Stream 型がある。Stream 型の要素は、任意個の unit 値を消費し、消費するたびに数と新たなストリームからなる二つ組を生成するような関数である。

```
Stream = μA. Unit→{Nat,A};
```

ストリームに対して二つの「分解子」を定義できる。s をストリームとすると、hd s は unit を渡したときに最初に返される数である。

```
hd = λs:Stream. (s unit).1;
▸ hd : Stream → Nat
```

同様に、tl s は unit を s に渡して得られる新たなストリームである。

```
    tl = λs:Stream. (s unit).2;
  ▸ tl : Stream → Stream
```

ストリームを構築するには、空腹関数の場合と同様に fix を用いる。

```
    upfrom0 = fix (λf: Nat→Stream. λn:Nat. λ_:Unit. {n,f (succ n)}) 0;
  ▸ upfrom0 : Stream

    hd upfrom0;
  ▸ 0 : Nat

    hd (tl (tl (tl upfrom0)));
  ▸ 3 : Nat
```

演習 20.1.2. [推奨, ★★] フィボナッチ数列 (1, 1, 2, 3, 5, 8, 13, ...) の一連の要素を生成するストリームを定義せよ。 □

ストリームをさらに一般化してある種の単純な**プロセス**にすることができる。プロセスは数を受け取り、数と新たなプロセスを返すような関数である。

```
    Process = μA. Nat→{Nat,A};
```

例えば、以下のプロセスは、毎回それまでに受け取った数の総和を返すようなプロセスである。

```
    p = fix (λf: Nat→Process. λacc:Nat. λn:Nat.
                let newacc = plus acc n in
                {newacc, f newacc}) 0;
  ▸ p : Process
```

ストリームの場合と同様に、プロセスとやり取りするための補助関数を定義できる。

```
    curr = λs:Process. (s 0).1;
  ▸ curr : Process → Nat

    send = λn:Nat. λs:Process. (s n).2;
  ▸ send : Nat → Process → Process
```

このプロセス p に数 5, 3, 20 を送ったとすると、最後のやり取りで返される数は 28 となる。

```
    curr (send 20 (send 3 (send 5 p)));
  ▸ 28 : Nat
```

オブジェクト

最後の例を少しだけ変更すると、データとやり取りするための別の馴染み深い方法、すなわちオブジェクトになる。例えば、数を保持し、問い合わせと増加を行えるカウンタのオブジェクトの型を以下に示す。

```
    Counter = μC. {get:Nat, inc:Unit→C};
```

ここでのオブジェクトの扱いは**純粋関数的**であることに注意されたい（これは第 19 章のオブジェクトと同様で、第 18 章のオブジェクトとは異なる）。カウンタのオブジェクトに inc メッセージを送っても、このオブジェクトの内部状態が変化するわけではない。その代わり、内部状態の値を増加させた**新しい**カウンタオブジェクトが返されるのである。ここで再帰型を使うことで、返り値オブジェクトの型を、元のオブジェクトと全く同じ型にすることができる。

これらのオブジェクトと上で議論したプロセスの唯一の違いは、オブジェクトが再帰的に定義された（関数を含む）**レコード**であるのに対して、プロセスは再帰的に定義された（組を返す）**関数**だということである。この視点の違いが便利なのは、レコードは複数の関数を含むように拡張できるためである。例えば、カウンタに減算の演算を加えられる。

```
Counter = μC. {get:Nat, inc:Unit→C, dec:Unit→C};
```

カウンタオブジェクトを生成するには、これまでと同様に不動点コンビネータを用いる。

```
c = let create = fix (λf: {x:Nat}→Counter. λs: {x:Nat}.
                        {get = s.x,
                         inc = λ_:Unit. f {x=succ(s.x)},
                         dec = λ_:Unit. f {x=pred(s.x)} })
      in create {x=0};
▸ c : Counter
```

c の演算を呼び出すには、単に適切なフィールドを射影し取り出せばよい。

```
c1 = c.inc unit;
c2 = c1.inc unit;
c2.get;
▸ 2 : Nat
```

演習 20.1.3. [★★] 上記の Counter 型とカウンタ c を拡張し、（18.7 節と同様に）backup と reset という演算を含むようにせよ。backup の呼び出しは、カウンタの現在の値を、別に用意された内部レジスタに保存する。reset の呼び出しは、そのレジスタの値でカウンタの値をリセットする。 □

再帰型を用いた再帰的な値の実現

再帰型のより驚くべき使い方で、その表現力を端的に示すものとして、不動点コンビネータの正しく型付けされた実装がある。任意の型 T に対して、T 上の関数のための不動点演算子は以下のようになる。

```
fixT = λf:T→T. (λx:(μA.A→T). f (x x)) (λx:(μA.A→T). f (x x));
▸ fixT : (T→T) → T
```

この項は、型を消去すれば、49 ページで見た型無しの不動点コンビネータと全く同じものであることに注意されたい[†3]。

鍵となる仕組みは、部分式 x x の二つの出現を、再帰型を用いて型付けしている点である。演習 9.3.2 で確認したように、この項を型付けするためには、x は関数型であって、かつその定義域の型は x の型そのものである必要がある。この性質を持つ有限の型は明らかに存在しないが、**無限の**型 μA.A→T はこの役割を完全に果たす。

この例の帰結として、再帰型の存在は強正規化可能性を壊す。fix_T コンビネータを用いることで、正しく型付けされた項で（unit に適用したときに）評価が発散（diverge）するものを書ける。

[†3] （訳注：原著正誤表より）この不動点コンビネータの型消去結果は、49 ページのものと完全に同じではない。ここでの定義には（値呼びの場合に機能するために必要な）y に関するラムダ抽象が欠けている。

```
  divergeT = λ_:Unit. fixT (λx:T. x);
▸ divergeT : Unit → T
```

さらに、任意の型に対して同様の項を得られるので、λ_ と異なり任意の型が要素を持つことがわかる[†4]。

型無しラムダ計算再考

もしかすると、再帰型の力を最も如実に示すのは、型無しラムダ計算の全体を、正しく型付けをしつつ、再帰型を持つ静的型付け言語に埋め込めるという事実かもしれない。D を以下の型とする[†5]。

```
  D = μX.X→X;
```

D から D への関数を D へと写す「入射関数」lam を以下のように定義する。

```
  lam = λf:D→D. f as D;
▸ lam : D
```

D のある要素を別の要素に適用（apply）するには、最初の要素の型を展開して関数を得、それを二つ目の要素に適用すればよい。

```
  ap = λf:D. λa:D. f a;
▸ ap : D
```

ここで M を、変数、ラムダ抽象、関数適用だけからなる閉じたラムダ項だとする。このとき、M を表す D の要素 M⋆ を、以下のように一様に構成できる。

```
  x⋆       =  x
  (λx.M)⋆  =  lam (λx:D. M⋆)
  (M N)⋆   =  ap M⋆ N⋆
```

例えば、以下は型無しの不動点コンビネータを D の要素として表したものである[†6]。

```
  fixD = lam (λf:D.
              ap (lam (λx:D. ap f (ap x x)))
                 (lam (λx:D. ap f (ap x x))));
▸ fixD : D
```

この純粋ラムダ計算の埋め込みは、数などの機能を含むよう拡張できる。D の定義をバリアント型に変えて、数のためのタグと、関数のためのタグを持つようにする。

```
  D = μX. <nat:Nat, fn:X→X>;
```

すなわち、D の要素は数であるか、もしくは D から D への関数であり、それぞれ nat と fn でタグ付けされる。lam 構築子の実装は本質的に前と同じままである。

```
  lam = λf:D→D. <fn=f> as D;
▸ lam : (D→D) → D
```

しかし、ap の実装には興味深い違いがある。

[†4] この事実により、再帰型を持つ体系は論理体系としては役に立たないものとなる。Curry-Howard 対応（9.4 節を見よ）に従って型を論理的な命題として解釈し、「型 T が要素を持つ」を「命題 T が証明可能」と読むと、任意の型が要素を持つという事実は、その論理のすべての命題が証明可能だということを意味する。すなわち、その論理は矛盾しているのである。

[†5] 表示的意味論に馴染みのある読者は、D の定義はまさに純粋なラムダ計算の意味モデルに使われる**普遍領域**を定義する性質そのものであることに気づくだろう。

[†6]（訳注：原著正誤表より）これは実際には名前呼びの不動点コンビネータ (Y) の実装であり、49 ページで定義した値呼びの不動点コンビネータではない。ただし、値呼びの不動点コンビネータも同様に再帰型を使って表現可能である。

```
  ap = λf:D. λa:D.
          case f of
            <nat=n> ⇒ divergeD unit
          | <fn=f> ⇒ f a;
▸ ap : D → D → D
```

f を a へと適用する前に、case を用いて f から関数を取り出す必要がある。これにより、f が関数で**ない**場合の適用の振る舞いを規定する必要が生じる。(この例では単に発散させているが、例外を発生させるという方法もある。) ここでのタグの検査が、Scheme などの動的型付け言語の実装で用いられる実行時のタグ検査に非常によく似ていることに注意されたい。この意味で、型付きの計算は、型無しの計算あるいは動的型付けの計算を「含む」といえるかもしれない。

D の要素に対する後者関数の定義でも同様のタグ検査が必要になる。

```
  suc = λf:D. case f of
                 <nat=n> ⇒ (<nat=succ n> as D)
               | <fn=f> ⇒ divergeD unit;
▸ suc : D → D
```

0 の D への入射は自明である。

```
  zro = <nat=0> as D;
▸ zro : D
```

演習 20.1.4. [★] この表現をブール値や条件式で拡張し、項 if false then 1 else 0 と項 if false then 1 else false とを D の要素として表現せよ。これらの項を評価したときには何が起こるだろうか。 □

演習 20.1.5. [推奨, ★★] データ型 D がレコードを含むよう拡張せよ。

```
  D = μX. <nat:Nat, fn:X→X, rcd:Nat→X>;
```

そして、レコードの構築とフィールドの射影を実装せよ。単純さのために、フィールドのラベルとしては自然数を用いる。すなわち、レコードは自然数から D の要素への関数として表現する。実装した拡張を検査器 fullequirec を用いてテストせよ。 □

20.2 形式的議論

型システムの文献では、再帰型への基本的なアプローチは二つに大別される。それらの間の本質的な違いは次の問いへの答えに凝縮されている。型 μX.T とそれを一段展開した型との間にある関係は何か? 例えば、NatList と <nil:Unit,cons:{Nat,NatList}> の関係は何か?

(1) **同値再帰**のアプローチでは、これら二つの型式は、両者が同じ無限の木を表しているため、**定義上等しい**とする。すなわち、任意の文脈で交換可能である[†7]。そして、一方を引数の型として期待する関数に、もう一方の型を持つ項を渡せることなどを保証するのは型検査器の責任である。

同値再帰の扱いの嬉しい点は、これまで理解してきた体系の宣言的な形式化に対して、型式が無限[†8]でもかまわないように変更するだけで済む点である。既存の定義、安全性定理、証

[†7] μ 型から無限木への写像は、236 ページで正確に定義する。
[†8] 厳密に言うなら、**正則** (21.7 節参照)。

→ μ　　　　　　　　　　　　　　　　　　　　　　　　$\lambda_{\rightarrow}$（図 9-1）の拡張

t ::=	...		項：
	fold [T] t		畳み込み
	unfold [T] t		展開
v ::=	...		値：
	fold [T] v		畳み込み
T ::=	...		型：
	X		型変数
	μX.T		再帰型

新しい評価規則　$\boxed{t \longrightarrow t'}$

$$\texttt{unfold [S] (fold [T] } v_1) \longrightarrow v_1 \qquad \text{(E-UnfldFld)}$$

$$\frac{t_1 \longrightarrow t_1'}{\texttt{fold [T] } t_1 \longrightarrow \texttt{fold [T] } t_1'} \qquad \text{(E-Fld)}$$

$$\frac{t_1 \longrightarrow t_1'}{\texttt{unfold [T] } t_1 \longrightarrow \texttt{unfold [T] } t_1'} \qquad \text{(E-Unfld)}$$

新しい型付け規則　$\boxed{\Gamma \vdash t : T}$

$$\frac{U = \mu X.T_1 \qquad \Gamma \vdash t_1 : [X \mapsto U]T_1}{\Gamma \vdash \texttt{fold [U] } t_1 : U} \qquad \text{(T-Fld)}$$

$$\frac{U = \mu X.T_1 \qquad \Gamma \vdash t_1 : U}{\Gamma \vdash \texttt{unfold [U] } t_1 : [X \mapsto U]T_1} \qquad \text{(T-Unfld)}$$

▲図 20-1.　同型再帰型（λμ）

明は、型式に関する帰納法に依存していない限り、変更しないでよい（型式に関する帰納法は当然そのままでは成立しなくなる）。

もちろん、型検査器は無限の構造を直接扱えないので、同値再帰型の**実装**には何らかの技術が要求される。それをどう実現するかが、まさに第 21 章の主題である。

(2) 一方、**同型再帰**のアプローチでは、再帰型とその展開結果は異なるが**同型**なものとして扱う。形式的には再帰型 μX.T の展開結果は、本体 T を取り、その中に現れる X をすべて再帰型全体で置き換えたものとなる。標準的な代入の記法を用いれば、これは [X ↦ (μX.T)]T となる。例えば、NatList 型の場合には

```
μX.<nil:Unit,cons:{Nat,X}>
```

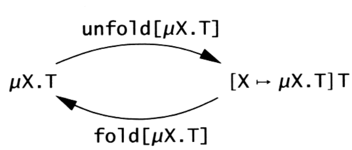
は以下へと展開される。

```
<nil:Unit, cons:{Nat, μX.<nil:Unit,cons:{Nat,X}>}>
```

同型再帰型を用いる体系では、各再帰型 μX.T に対して、以下の関数の組を導入する。

```
unfold[μX.T]   :  μX.T → [X ↦ μX.T]T
fold[μX.T]     :  [X ↦ μX.T]T → μX.T
```

これらの関数は二つの型の間で値を相互に写すことで「同型性の証明を与える」。



写像 fold と写像 unfold は、図 20-1 に示すように、言語によってプリミティブとして提供される。これらが同型写像をなすという事実は評価規則 E-UnfldFld によって捉えられる。

この規則によって、fold が対応する unfold と合わさると、fold が打ち消される。(この評価規則は、fold と unfold の型注釈が同一であることを要求していない。というのは、そのような制約を検証するには、実行時に型検査器を呼び出さなくてはならないためである。しかし、正しく型付けされたプログラムの評価では、E-UnfldFld が適用されるときには常に二つの型注釈は同一に**なるはず**である。)

これらのアプローチは共に理論的研究とプログラミング言語の設計の両方で用いられている。同値再帰のスタイルはほぼ間違いなく直観的だが、型検査器への要求は大きくなり、型検査器は実質的に fold と unfold の注釈がどこで必要かを推論しなくてはならない。さらに、同値再帰型と、他の発展的な型付け機能(有界量化や型演算子など)との組み合わせは非常に複雑になりうる。それにより、理論的にも非常に困難になったり(Ghelli [172]; Colazzo and Ghelli [104] など)、型検査問題が決定不能になってしまうことすらある(Solomon [437])。

同型再帰のスタイルは記法的にはいくぶん重厚で、再帰型を用いる際には常に、fold と unfold の指示をプログラムに書き加えないといけない。しかし実用上は、これらの注釈は他の注釈との合成で「隠蔽される」ことが多い。例えば、ML 系の言語では、datatype 定義は常に再帰型を暗黙に導入する。いずれかの構築子を使ってこのデータ型の値を構築するときは常に、暗黙に fold が呼ばれる。パターンマッチ中に現れる構築子は暗黙に unfold を強制する。同様に、Java の各クラス定義は暗黙に再帰型を導入し、オブジェクトのメソッド呼び出しは暗黙に unfold を呼び出す。このように機構をうまく重ね合わせることによって、同型再帰のスタイルは実用上は非常に快適なものになる。

例として、同型再帰の形式で NatList 型の定義を行う。まず簡便のため、NatList の展開結果のための略記を以下のように定義しておく。

```
NLBody = <nil:Unit, cons:{Nat,NatList}>;
```

すると nil は、NLBody 型のバリアントを作り、それを NatList に畳み込むものとして定義される。cons についても同様である。

```
nil = fold [NatList] (<nil=unit> as NLBody);
cons = λn:Nat. λl:NatList. fold [NatList] <cons={n,l}> as NLBody;
```

逆に、isnil, hd, tl の定義では、NatList を受け取り、それをバリアントとみなすことでタグによる case 分岐ができるようにする。これは引数 l を展開することで実現できる。

```
isnil = λl:NatList.
          case unfold [NatList] l of
            <nil=u> ⇒ true
          | <cons=p> ⇒ false;
hd = λl:NatList.
        case unfold [NatList] l of
          <nil=u> ⇒ 0
        | <cons=p> ⇒ p.1;
tl = λl:NatList.
        case unfold [NatList] l of
          <nil=u> ⇒ l
        | <cons=p> ⇒ p.2;
```

演習 20.2.1. [推奨, ★★] 20.1 節にある他の例の幾つか(特に、213 ページの $\mathtt{fix}_T$ の例)を、明示的に fold と unfold を用いて定式化し直せ。結果は fullisorec 検査器を用いて確認せよ。

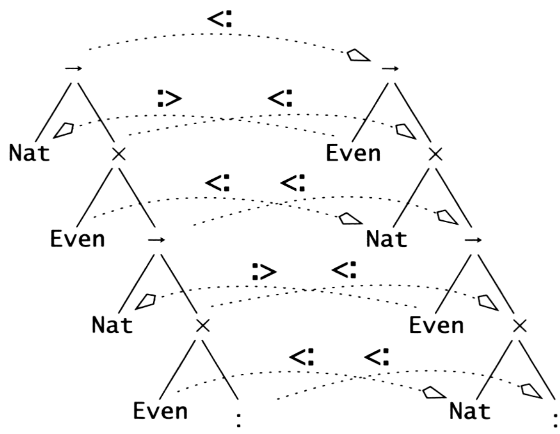
□

演習 20.2.2. [★★ ↛] 同型再帰型システムのための進行定理と保存定理の証明の概略を示せ。 □

20.3 部分型付け

この章で解決しておかなければならない最後の課題は、これまでに登場した単純型付きラムダ計算のもう一つの大きな改良、すなわち部分型付けを、再帰型と組み合わせることである。例えば、Even を Nat の部分型としよう。このとき、型 μX.Nat→(Even×X) と μX.Even→(Nat×X) の間の関係はどうあるべきだろうか。

最も単純な考え方は、これらを「極限において」考える、すなわち再帰型を同値再帰として扱うというものである。この例では、どちらの型の要素も単純なプロセス（212 ページ参照）とみなせる。数が与えられると、別の数と、数を受け取れる状態の新たなプロセスを返す、ということを繰り返す。最初の型に属するプロセスは常に偶数を返し、任意の数を受け取る能力を持つ。二つ目の型に属するプロセスは、任意の数を生成しうるが、受け取る数としては常に偶数を想定している。受け取るべき引数に関する関数への制約と、許される返り値に関する関数への制約は、どちらも前者の型のほうが厳しい要求になっている。したがって、直観的には前者は後者の部分型であることが期待される。これらの計算をまとめたのが以下の図である。



この直観的な議論を正確なものにできるだろうか。もちろん可能で、第 21 章で実際にそれを行う。

20.4 注記

計算機科学において再帰型の歴史は少なくとも Morris [326] にまで遡る。部分型付けを除いた再帰型の基本的な構文的性質と意味論的性質は、Huet [228] および MacQueen, Plotkin, and Sethi [295] らの初期の論文で証明され、Cardone and Coppo [90] にまとめられている。（さらに初期の Claude Pair の論文 “*Concerning the Syntax of Algol 68*” には、同値再帰に関する等号の決定可能性の最初の証明と思われるものが含まれている。Algol Bulletin 誌（no. 31, March 1970）に掲載された原本を見つけるのは難しいが、Pierre Lescanne は親切にも彼のウェブサイトでスキャンしたものを閲覧できるようにしてくれている[†9]。） 無限かつ正則な木の性質については、Courcelle [119] がサーベイを行っている。

---

†9 (訳注) http://www.ens-lyon.fr/LIP/REWRITING/TYPES_AND_L_CALCULUS/RecursiveTypes/

再帰型を使えば正しく型付けされた `fix` 演算子を構成できる、ということに最初に気づいたのは Morris [326]（pp. 122–124）である（213 ページ参照)。

同型再帰型の型システムと同値再帰型の型システムの関係は、Abadi and Fiore [10] によって研究された。再帰型の二つの形式化は、この分野の初期の研究からずっと存在していた。しかし、「**同型再帰**」「**同値再帰**」という覚えやすい用語は、最近になって Crary, Harper, and Puri [123] によって考案されたものである。

さらに、部分型付けを伴う再帰型に関する参考文献が 21.12 節にある。

第 21 章

再帰型のメタ理論

本章で扱う体系は、部分型付け（図 15-1）、直積型（図 11-5）、同値再帰を伴う単純型付き計算である。対応する検査器は equirec である。

第 20 章では、再帰型の異なった二つの表し方として、展開結果と定義上等価な同値再帰型と、等価性を項 `fold` および `unfold` によって明示的に示す同型再帰型を扱った。この章では、同値再帰型の型検査器の理論的基礎を与える（同型再帰型の実装は比較的単純である）。再帰型と部分型付けは実際にはしばしば組み合わせて使うため、それらを両方含んだ型システムを扱うことにする。同値再帰型だけを扱い、部分型付けを扱わない型システムを考えても、さほど単純にはならない。というのは、再帰型の等価性の検査がいずれにせよ必要なためである。

第 20 章では、同値再帰型の部分型付けは直観的には、無限型上の部分型付けの無限の導出として考えられることを見てきた。ここでは、これらの直観を**余帰納法**という数学的枠組みを用いて厳密化し、無限の木や導出と、実際の部分型付けアルゴリズムが操作する有限の表現の間の厳密な関係を導く。

まず、21.1 節で帰納的定義と余帰納的定義およびそれらに結びついた証明原理の基本的な理論を概観する。次に、21.2 節と 21.3 節では、この一般的な理論を部分型付けの場合に適用し、馴染みのある有限型に関する帰納的な部分型関係と、その無限型への余帰納的な拡張の両方を定義する。21.4 節では少し回り道をし、推移律の規則に関する問題を幾つか考える（この規則は、これまで見てきたように、部分型付けを持つ体系のトラブルメーカーとして悪名高い）。21.5 節では帰納的もしくは余帰納的に定義された集合の所属関係を検査する単純なアルゴリズムを導出し、21.6 節ではより改良されたアルゴリズムについて考える。21.7 節 では、「正則」な無限型という重要かつ特別な場合の部分型付けに対して、これらのアルゴリズムを適用する。21.8 節では無限型を表す有限の記法として μ 型を導入し、μ 型上のより複雑な（だが有限で実装可能な）部分型関係が、無限型の間の部分型関係の通常の余帰納的定義に対応することを証明する。21.9 節では μ 型の部分型付けアルゴリズムの停止性を証明する。21.10 節ではこのアルゴリズムを、Amadio と Cardelli による別のアルゴリズムと比較する。21.11 節 では同型再帰型について簡潔に論じる。

21.1 帰納法と余帰納法

普遍集合 $\mathcal{U}$ を固定し、この領域上で帰納的定義と余帰納的定義の議論を展開することにしよう。$\mathcal{U}$ は「森羅万象」からなる集合を表し、帰納的定義や余帰納的定義の役割は $\mathcal{U}$ の部分集合を選び出すことである。（後で、すべての型の二つ組の集合を $\mathcal{U}$ として選ぶ。この場合、$\mathcal{U}$ の部分集合は型の上の関係となる。差し当たっては $\mathcal{U}$ はどんな集合でもよい。）

定義 21.1.1. 関数 $F \in \mathcal{P}(\mathcal{U}) \to \mathcal{P}(\mathcal{U})$ は、$X \subseteq Y$ ならば $F(X) \subseteq F(Y)$ であるとき、**単調**であるという。（$\mathcal{P}(\mathcal{U})$ が $\mathcal{U}$ のすべての部分集合の集合であることを思い出そう。） □

以下では、F が $\mathcal{P}(\mathcal{U})$ 上の何らかの単調関数であると仮定する。F のことをしばしば**生成関数**と呼ぶ。

定義 21.1.2. X を $\mathcal{U}$ の部分集合とする。

(1) $F(X) \subseteq X$ であるときに、X は「F について**閉じている**」という。
(2) $X \subseteq F(X)$ であるときに、X は「F について**整合的**である」という。
(3) $F(X) = X$ であるときに、X は F の**不動点**であるという。 □

これらの定義を理解する上で有用な直観は、$\mathcal{U}$ の要素を言明や表明であると考え、F を「正当化」関係、すなわち与えられた言明の集合（前提）に対し、そこから導かれる新たな言明（結論）を対応付けるような関係と考えることである。F について閉じている集合は、F によって正当化される要素を加えても大きくすることのできない集合である。F について閉じている集合は、その要素によって正当化されるような結論を既にすべて含む。一方、F について整合的な集合は、「自己正当化」している集合である。すなわち、F について整合的な集合に属する言明はすべて、同じ集合に属する他の言明によって、正当化されている。F の不動点は閉じていてかつ整合的である集合である。すなわち、要素が必要とする正当化と、要素から導かれる結論をすべて含み、それ以外は含まない。

例 21.1.3. 三要素の普遍集合 $\mathcal{U} = \{a, b, c\}$ 上の以下の生成関数を考えよう。

$$\begin{array}{lcllcl} E_1(\emptyset) & = & \{c\} & E_1(\{a,b\}) & = & \{c\} \\ E_1(\{a\}) & = & \{c\} & E_1(\{a,c\}) & = & \{b,c\} \\ E_1(\{b\}) & = & \{c\} & E_1(\{b,c\}) & = & \{a,b,c\} \\ E_1(\{c\}) & = & \{b,c\} & E_1(\{a,b,c\}) & = & \{a,b,c\} \end{array}$$

このとき、E_1 について閉じている集合はただ一つ $\{a,b,c\}$ だけしか存在せず、E_1 について整合的な集合は、$\emptyset$、$\{c\}$、$\{b,c\}$、$\{a,b,c\}$ の四つが存在する。

E_1 は以下のような**推論規則**の集まりとして簡潔に表現できる。

$$\frac{}{c} \qquad \frac{c}{b} \qquad \frac{b \quad c}{a}$$

各規則は横線の上の要素がすべて入力の集合に含まれるときに、横線の下の要素が出力の集合に含まれるということを表している。 □

定理 21.1.4.［**Knaster-Tarski（Tarski, 1955 [456]）**］

(1) F について閉じているすべての集合の共通部分は F の最小不動点である。
(2) F について整合的なすべての集合の和集合は F の最大不動点である。 □

［証明］　ここでは (2) のみを示す。(1) の証明はそれと対称的な証明である。$C = \{X \mid X \subseteq F(X)\}$ を F について整合的な集合すべての集まりとし、それら集合の和集合を P とする。F が単調であるという事実と、任意の $X \in C$ について X が F について整合的かつ $X \subseteq P$ であることがわかっていることを踏まえると、$X \subseteq F(X) \subseteq F(P)$ がいえる。結果として、$P = \bigcup_{X \in C} X \subseteq F(P)$ すなわち P が F について整合的であることがいえる。さらに、定義により P は F について整合的な最大の集合である。F の単調性を再度用いると、$F(P) \subseteq F(F(P))$ がいえる。ここから、C の定義より $F(P) \in C$ がいえる。したがって、C の任意の要素と同様に、$F(P) \subseteq P$ すなわち P が F について閉じていることがいえる。ここまでで P が F について整合的な最大の集合であることと、P が F の不動点であることがいえたので、P は最大不動点である。 □

定義 21.1.5. F の最小不動点は μF で表し、F の最大不動点は νF で表す。 □

例 21.1.6. 上で例として用いた生成関数 E_1 の場合、$\mu E_1 = \nu E_1 = \{a,b,c\}$ となる。 □

演習 21.1.7.［★］　普遍集合 $\{a,b,c\}$ 上の以下の生成関数 E_2 が以下の推論規則で定義されているとする。

$$\frac{}{a} \qquad \frac{c}{b} \qquad \frac{a \quad b}{c}$$

上の E_1 でしたように、関係 E_2 に含まれる組の集合を明示的に書き下せ。E_2 について閉じている集合と E_2 について整合的な集合をすべて列挙せよ。μE_2 と νE_2 はどうなるか。 □

μF それ自体 F について閉じている（よって F について閉じている最小の集合である）こととと、νF も F について整合的である（よって F について整合的な最大の集合である）ことに注意されたい。この観察から、二つの基礎的な推論の道具が得られる。

系 21.1.8.［定理 21.1.4 の系］

(1) **帰納法原理：** X が F について閉じているならば、$\mu F \subseteq X$ である。
(2) **余帰納法原理：** X が F について整合的ならば、$X \subseteq \nu F$ である。 □

これらの原理の背景となる直観は、X を述語の特性集合（述語が真となる $\mathcal{U}$ の部分集合）による表現とみなす考え方からきている。この考え方では、要素 x が性質 X を満たすことを示すのは、x が集合 X の要素であることを示すのと同じである。そうすると、帰納法原理がいっているのは、性質の特性集合が F に関して閉じている（すなわち、その性質が F で保たれる）とき、帰納的に定義された集合 μF のすべての要素に対してその性質が成り立つということである。

一方、余帰納法原理は、要素 x が余帰納的に定義された集合 νF の**要素である**ことを証明するための手段を提供している。$x \in \nu F$ を示すには、集合 X で $x \in X$ かつ F について整合的であるものを見つけることができればよい。帰納法原理ほどには知られていないが、余帰納法原理は計算機科学の多くの分野で中心的役割を果たしている。例えば、**双模倣性**に基づいた並行性の理論での主要な証明技法であり、多くの**モデル検査**アルゴリズムの心臓部でもある。

帰納法原理と余帰納法原理は、この章全体にわたって頻繁に用いられている。簡潔さのため、帰納法を用いた議論をすべて生成関数と述語を用いて行うのではなく、構造的帰納法といった省略形に頼ることも多い。余帰納法を用いた議論については、より明示的に述べる。

演習 21.1.9.［推奨, ★★★］ 自然数上の通常の帰納法の原理（公理 2.4.1）と、自然数の二つ組上の辞書式帰納法の原理（公理 2.4.4）が、系 21.1.8 の帰納法原理から導かれることを示せ。 □

21.2 有限型と無限型

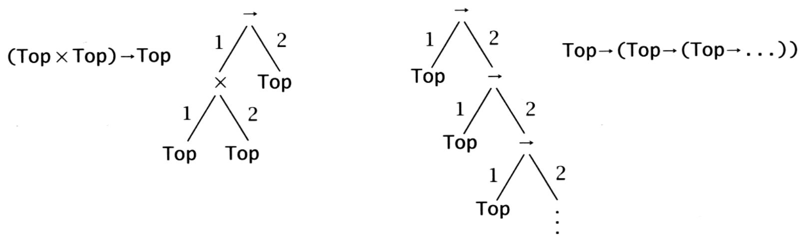
最大不動点と余帰納的な証明法の一般的な定義を、部分型付けの場合へと具体化していく。ただしその前に、型をどのように（有限もしくは無限の）木とみなすか厳密に説明しておく必要がある。

簡潔さのため、この章では →, ×, Top の三つの型構築子のみを扱う。型を（無限かもしれない）木で、各ノードが →, ×, Top のいずれかをラベルとして持つようなものとして表す。この定義はこの章で必要なものに特化したものになっている。無限のラベル付き木の一般的な取り扱いについては Courcelle [119] を参照せよ。

1 と 2 からなる列からなる集合を $\{1,2\}^*$ と書く。また、空列を $\bullet$ と書き、i の k 個の繰り返しを i^k を書くことを思い出そう。π と σ が列のとき、π,σ で π と σ の連結を表す。

定義 21.2.1. **木型**[†1]（もしくは単に木）とは、以下の制約を満たす部分関数 $\mathsf{T} \in \{1,2\}^* \rightharpoonup \{\to, \times, \mathsf{Top}\}$ である。

[†1] 「木型」という言い回しは若干不格好だが、21.8 節で再帰型を μ を含む有限の式（「μ 型」）として表す際には、物事をきちんとしておく助けとなる。



▲ 図 21-1. 木型の例

- T(•) は定義されている。
- T(π,σ) が定義されているならば、T(π) も定義されている。
- T(π) = → もしくは T(π) = × ならば、T(π,1) と T(π,2) は定義されている。
- T(π) = Top ならば、T(π,1) と T(π,2) は未定義である。

dom(T) が有限であるときに、木型 T は**有限**（finite）であるという。すべての木型の集合を $\mathcal{T}$ と表し、すべての有限の木型からなるその部分集合を $\mathcal{T}_f$ と表す。 □

記法の便宜上、T(•) = Top である木 T を Top と書くことにする。また、T_1 と T_2 が木であるとき、$T_1 \times T_2$ で $(T_1 \times T_2)(\bullet) = \times$ かつ $(T_1 \times T_2)(i,\pi) = T_i(\pi)$ である木を表し、$T_1 \to T_2$ で $(T_1 \to T_2)(\bullet) = \to$ かつ $(T_1 \to T_2)(i,\pi) = T_i(\pi)$ である木を表す（ただし $i = 1,2$）。例えば、(Top × Top)→Top は関数 T(•) = →、T(1) = ×、T(2) = T(1,1) = T(1,2) = Top で定義される有限の木型を表す。非形式的な記法として、「...」で有限でない木型を表す。例えば、Top→(Top→(Top→...)) は、すべての $k \geq 0$ について $T(2^k) = \to$ かつ、すべての $k \geq 0$ について $T(2^k,1) = \mathsf{Top}$ という関数によって定義される型 T に対応する。図 21-1 はこれらの表記の説明である。

有限の木型の集合は以下の文法を用いてよりコンパクトに定義することもできる。

```
T  ::=  Top
        T×T
        T→T
```

形式的には、$\mathcal{T}_f$ はこの文法規則で表された生成関数の最小不動点である。この生成関数の普遍集合は、Top, →, × でラベル付けされたすべての有限および無限の木である（すなわち定義 21.2.1 から最後の二つの条件を除いて一般化することで得られる集合である）。$\mathcal{T}$ 全体は、同じ生成関数から最小不動点の代わりに最大不動点を取ることで得られる。

演習 21.2.2. [推奨, ★★] 前の段落の考えに基づいて、普遍集合 $\mathcal{U}$ と生成関数 $F \in \mathcal{P}(\mathcal{U}) \to \mathcal{P}(\mathcal{U})$ で、有限の木型の集合 $\mathcal{T}_f$ が F の最小不動点になり、すべての木型の集合 $\mathcal{T}$ が最大不動点になるものを述べよ。 □

21.3 部分型関係

有限木型上の部分型関係と、木型一般の上の部分型関係を、適切な普遍集合上の単調関数の最小不動点および最大不動点としてそれぞれ定義する。有限木型上の部分型関係の場合、有限木型の組からなる集合 $\mathcal{T}_f \times \mathcal{T}_f$ を普遍集合として用いる。生成関数はこの普遍集合の部分集合（すなわち $\mathcal{T}_f$

上の関係）を他の部分集合へと写し、生成関数の不動点も同様に $\mathcal{T}_f$ 上の関係となる。任意の（有限もしくは無限の）木型の場合、普遍集合は $\mathcal{T} \times \mathcal{T}$ である。

定義 21.3.1. ［有限の部分型関係］ 二つの有限木型 S と T は、$(\mathsf{S},\mathsf{T}) \in \mu S_f$ であるとき、部分型関係にある（「S が T の部分型である」）という。ここで、単調関数 $S_f \in \mathcal{P}(\mathcal{T}_f \times \mathcal{T}_f) \to \mathcal{P}(\mathcal{T}_f \times \mathcal{T}_f)$ は以下で定義される関数である。

$$
\begin{array}{rcl}
S_f(R) & = & \{(\mathsf{T},\mathsf{Top}) \mid \mathsf{T} \in \mathcal{T}_f\} \\
& \cup & \{(\mathsf{S_1}\times\mathsf{S_2}, \mathsf{T_1}\times\mathsf{T_2}) \mid (\mathsf{S_1},\mathsf{T_1}),\ (\mathsf{S_2},\mathsf{T_2}) \in R\} \\
& \cup & \{(\mathsf{S_1}\to\mathsf{S_2}, \mathsf{T_1}\to\mathsf{T_2}) \mid (\mathsf{T_1},\mathsf{S_1}),\ (\mathsf{S_2},\mathsf{T_2}) \in R\}
\end{array}
$$

この生成関数は以下の推論規則の集まりによる部分型関係の標準的定義の働きを正確に捉えている。

$$\frac{}{\mathsf{T} <: \mathsf{Top}}$$

$$\frac{\mathsf{S_1} <: \mathsf{T_1} \qquad \mathsf{S_2} <: \mathsf{T_2}}{\mathsf{S_1}\times\mathsf{S_2} <: \mathsf{T_1}\times\mathsf{T_2}}$$

$$\frac{\mathsf{T_1} <: \mathsf{S_1} \qquad \mathsf{S_2} <: \mathsf{T_2}}{\mathsf{S_1}\to\mathsf{S_2} <: \mathsf{T_1}\to\mathsf{T_2}}$$

二つ目と三つ目の規則での、線の上の S <: T という部分型付け判断式は「もし二つ組 (S, T) が S_f の引数に含まれるならば」と読み、線の下の S <: T という部分型付け判断式は「そのとき (S, T) は S_f の結果に含まれる」と読む。 □

定義 21.3.2. ［無限の部分型関係］ 二つの（有限もしくは無限の）木型 S と T は、$(\mathsf{S},\mathsf{T}) \in \nu S$ であるときに部分型関係にあるという。ここで $S \in \mathcal{P}(\mathcal{T} \times \mathcal{T}) \to \mathcal{P}(\mathcal{T} \times \mathcal{T})$ は以下で定義される関数である。

$$
\begin{array}{rcl}
S(R) & = & \{(\mathsf{T},\mathsf{Top}) \mid \mathsf{T} \in \mathcal{T}\} \\
& \cup & \{(\mathsf{S_1}\times\mathsf{S_2}, \mathsf{T_1}\times\mathsf{T_2}) \mid (\mathsf{S_1},\mathsf{T_1}),\ (\mathsf{S_2},\mathsf{T_2}) \in R\} \\
& \cup & \{(\mathsf{S_1}\to\mathsf{S_2}, \mathsf{T_1}\to\mathsf{T_2}) \mid (\mathsf{T_1},\mathsf{S_1}),\ (\mathsf{S_2},\mathsf{T_2}) \in R\}
\end{array}
$$

この関係を推論規則で表したものは、上で定義した帰納的関係の場合と全く同じであることに注意されたい。変わったのは、型のより大きな普遍集合を考えていることと、最小不動点の代わりに最大不動点を取っていることだけである。 □

演習 21.3.3. ［★］ νS に属さない二つ組 (S, T) を示すことで、νS が $\mathcal{T} \times \mathcal{T}$ 全体でないことを確かめよ。 □

演習 21.3.4. ［★］ νS によって関連付けられるが、μS によっては関連付けられないような型の二つ組 (S, T) はあるか。また、μS_f によって関連付けられるが、νS_f によっては関連付けられないような型の二つ組 (S, T) についてはどうか。 □

無限木の部分型関係が推移的であるという基本的性質は、直ちに確認すべきである。（有限型の部分型関係が推移的であることは 16.1 節で確認済みである。）もし仮に部分型関係が推移的で**ない**としたら、評価過程での型の保存という重大な性質が直ちに失われてしまう。これを確認するために、型 S, T, U で S<:T と T<:U は成り立つが S<:U が成り立たないものがもしあったとしよう。s を S 型の値とし、f を U→Top 型の関数としよう。このとき、項 (λx:T. f x) s は関数適用のそれぞれに対して一回ずつ包摂規則を用いることで型付けできる。しかし、1 ステップ簡約すると、正しく型付けされない項 f s になってしまう。

定義 21.3.5. 関係 $R \subseteq \mathcal{U} \times \mathcal{U}$ は、次の単調関数 $TR(R) = \{(x, y) \mid \exists z \in \mathcal{U}. (x, z), (z, y) \in R\}$ に関して閉じている、すなわち $TR(R) \subseteq R$ であるときに、**推移的**であるという。 □

補題 21.3.6. $F \in \mathcal{P}(\mathcal{U} \times \mathcal{U}) \to \mathcal{P}(\mathcal{U} \times \mathcal{U})$ を単調関数とする。任意の $R \subseteq \mathcal{U} \times \mathcal{U}$ について、$TR(F(R)) \subseteq F(TR(R))$ ならば、νF は推移的である。 □

[証明] νF は不動点であるので $\nu F = F(\nu F)$ であり、$TR(\nu F) = TR(F(\nu F))$ がいえる。したがって、この補題の仮定より $TR(\nu F) \subseteq F(TR(\nu F))$ である。言い換えると、$TR(\nu F)$ は F について整合的であるので、余帰納法の原理より $TR(\nu F) \subseteq \nu F$ である。定義 21.3.5 により、これは νF が推移的であることと同値である。 □

この補題は一般に推論体系において推移律の規則が冗長であることを確立するための伝統的な手法（しばしば「カット除去の証明」と呼ばれる。16.1 節を参照）を彷彿とさせる。条件 $TR(F(R)) \subseteq F(TR(R))$ は、この手法の非常に重要な部分に対応している。つまり、ある言明が、R の幾つかの言明を取り、それに対して F の幾つかの規則を適用し、推移律の規則 TR を適用して得られるとき、これらの順序を逆転させ、最初に推移律を適用し、その後 F の規則を適用しても、同じ言明が得られると論じるのである。この補題を用いて、部分型関係の推移律を証明する。

定理 21.3.7. νS は推移的である。 □

[証明] 補題 21.3.6 により、任意の $R \subseteq \mathcal{T} \times \mathcal{T}$ に対して、$TR(S(R)) \subseteq S(TR(R))$ であることを示せば十分である。$(\mathsf{S}, \mathsf{T}) \in TR(S(R))$ とする。TR の定義より、ある $\mathsf{U} \in \mathcal{T}$ が存在して、$(\mathsf{S}, \mathsf{U}), (\mathsf{U}, \mathsf{T}) \in S(R)$ を満たす。証明のゴールは、$(\mathsf{S}, \mathsf{T}) \in S(TR(R))$ であることを示すことである。U の形の可能性を場合分けして考える。

$\mathsf{U} = \mathsf{Top}$ の場合

$(\mathsf{U}, \mathsf{T}) \in S(R)$ であるので、S の定義により、T は Top でなくてはならない。しかし、任意の A と Q について、$(\mathsf{A}, \mathsf{Top}) \in S(Q)$ であり、特に $(\mathsf{S}, \mathsf{T}) = (\mathsf{S}, \mathsf{Top}) \in S(TR(R))$ である。

$\mathsf{U} = \mathsf{U}_1 \times \mathsf{U}_2$ の場合

もし、$\mathsf{T} = \mathsf{Top}$ ならば、前の場合と同様に $(\mathsf{S}, \mathsf{T}) \in S(TR(R))$ である。そうでない場合、$(\mathsf{U}, \mathsf{T}) \in S(R)$ から、$\mathsf{T} = \mathsf{T}_1 \times \mathsf{T}_2$ かつ $(\mathsf{U}_1, \mathsf{T}_1), (\mathsf{U}_2, \mathsf{T}_2) \in R$ である。同様に、$(\mathsf{S}, \mathsf{U}) \in S(R)$ から、$\mathsf{S} = \mathsf{S}_1 \times \mathsf{S}_2$ かつ $(\mathsf{S}_1, \mathsf{U}_1), (\mathsf{S}_2, \mathsf{U}_2) \in R$ である。TR の定義により $(\mathsf{S}_1, \mathsf{T}_1), (\mathsf{S}_2, \mathsf{T}_2) \in TR(R)$ であり、ここから S の定義により $(\mathsf{S}_1 \times \mathsf{S}_2, \mathsf{T}_1 \times \mathsf{T}_2) \in S(TR(R))$ がいえる。

$\mathsf{U} = \mathsf{U}_1 \to \mathsf{U}_2$ の場合

同様。 □

演習 21.3.8. [推奨, ★★] 無限木型上の部分型関係が反射的でもあることを示せ。 □

以下の節では推移律についての議論を継続し、有限型の部分型関係の標準的な説明における推移律の取り扱いと、無限木型の部分型関係の今の説明における推移律の取り扱いとを比較する。初読の際には飛ばしたり流し読みしたりしてよい。

21.4 推移律に関する余談

第 16 章で、部分型関係の帰納的定義には二つの標準的な形式化の形があるのを見た。一つは**宣言的**なもので、これは読みやすさに特化している。もう一つは、**アルゴリズム的**なもので、多かれ少なかれ実装に直結している。単純な体系では、これらの二つの形式はほとんど同じだが、より複雑な体系では大きく異なるようになり、これらが型上の同じ関係を定義していることを証明するのは大きな課題となる。（この例を第 28 章で確認する。また他の多くの例が研究されてきた。）

宣言的な表現とアルゴリズム的な表現の最も特徴的な違いの一つに、宣言的な表現が推移律の規則（S<:U かつ U<:T ならば S<:T）を明示的に含んでいるのに対して、アルゴリズム的な表現はそうではないという点がある。ゴール駆動でこの規則を適用するには U の推測が必要なため、この規則はアルゴリズムの役には立たないのである。

推移律の規則は宣言的な型システムで二つの有用な役割を果たしている。第一に、部分型関係がまさに推移的であることを読者に明らかにしている。第二に、推移律の規則によって、他の規則をより単純かつ基本的な形で表現することが可能になっている。アルゴリズム的な表現では、そういった単純な規則は、それら規則のあらゆる組み合わせを考慮に入れた、重厚で巨大な規則へと統合しなくてはならない。例えば、推移律の規則が存在する場合には、レコードのフィールドの「深さ部分型付け」、フィールドの追加による「幅部分型付け」、フィールドの「並べ替え」は、15.2 節でしたように独立して表現することができ、これらすべてをよりわかりやすくできる。推移律の規則なしには、これら三つの規則は、16.1 節でしたように、幅、深さ、並べ替えのすべてを同時に考慮に入れた単一の規則にしなくてはならない。

いくぶん驚くべきことだが、推移律の規則を用いた宣言的な表現を与えられるのは、帰納的な定義で使えるが余帰納的な定義では使えない「トリック」の帰結であることがわかる。これを理解するために、推移律が**閉包性**の一種であることに注目せよ。すなわち、部分型関係が推移律の規則に関して閉じていることを要請しているのである。有限型上の部分型関係はそれ自体が規則の集合の閉包として定義されるので、推移律の元で閉じるようにするためには、単に推移律を規則の集合に加えればよい。規則の二つの集合の和集合を帰納的に適用することで、両方の規則の集合のそれぞれについて閉じた最小の集合が生成される。これは帰納的定義と閉包性の一般的な性質である。この事実は生成関数を用いて以下のようにより抽象的に形式化できる。

命題 21.4.1. F と G を単調な関数とし、$H(X) = F(X) \cup G(X)$ とする。このとき、μH は F と G の両方について閉じている最小の集合である。 □

［証明］ まず最初に μH が F と G の両方について閉じていることを示す。定義により、$\mu H = H(\mu H) = F(\mu H) \cup G(\mu H)$ であるので、$F(\mu H) \subseteq \mu H$ かつ $G(\mu H) \subseteq \mu H$ である。次に、μH が F と G の両方について閉じた**最小**の集合であることを示す。$F(X) \subseteq X$ かつ $G(X) \subseteq X$ であるような集合 X があったとする。このとき $H(X) = F(X) \cup G(X) \subseteq X$ であり、すなわち X は H について閉じている。（Knaster-Tarski の定理により）μH は H について閉じた最小の集合なので、$\mu H \subseteq X$ がいえる。 □

残念ながら、推移閉包を実現するこのトリックは、余帰納的な定義を扱っているときには、使うことができない。以下の演習が示すように、余帰納的な定義の生成規則に推移律の規則を追加すると、常に縮退した関係になってしまうのである。

演習 21.4.2. ［★］ F を普遍集合 $\mathcal{U}$ 上の生成関数とする。生成関数

$$F^{TR}(R) = F(R) \cup TR(R)$$

の最大不動点 νF^{TR} が、$\mathcal{U} \times \mathcal{U}$ 上の「全関係」(すべての元と元が関係しているような関係) であることを示せ。 □

よって、余帰納的な場合には、宣言的な表現はやめ、アルゴリズム的な表現だけを扱うことにする。

21.5 所属性検査

それでは、この章の中心的な問いへと注意を向けよう。すなわち、ある普遍集合 $\mathcal{U}$ 上の生成関数 F と要素 $x \in \mathcal{U}$ が与えられたときに、x が F の最大不動点に含まれるか否かを決定する方法である。最小不動点の場合の所属性検査については、より簡潔に (演習 21.5.13 で) 取り組む。

与えられた要素 $x \in \mathcal{U}$ を F で生成する方法は、一般には複数ありうる。すなわち、集合 $X \subseteq \mathcal{U}$ で $x \in F(X)$ であるようなものは、複数ありうるのである。そのような集合 X を x の**生成集合**と呼ぶ。F の単調性から、x の生成集合の上位集合はすべて x の生成集合であるので、極小な生成集合に限定して注目していくのが有意義である。これをさらに一歩進め、「可逆」な生成関数、すなわち x が高々一つの極小な生成集合しか持たないような生成関数のクラスに注目できる。

定義 21.5.1. 生成関数 F は、任意の $x \in \mathcal{U}$ について、集合の集まり

$$G_x = \{X \subseteq \mathcal{U} \mid x \in F(X)\}$$

が空であるか、もしくは他のすべての要素の部分集合である一意な要素を含むかであるとき、**可逆**であるという。F が可逆であるとき、部分関数 $support_F \in \mathcal{U} \to \mathcal{P}(\mathcal{U})$ を以下のように定義する。これを支持関数 (support function) と呼ぶ[†2]。

$$support_F(x) = \begin{cases} X & X \in G_x \text{ かつ } \forall X' \in G_x.\ X \subseteq X' \text{ の場合} \\ \uparrow & G_x = \varnothing \text{ の場合} \end{cases}$$

支持関数は以下のように集合に対しても定義しておく。

$$support_F(X) = \begin{cases} \bigcup_{x \in X} support_F(x) & \forall x \in X.\ support_F(x)\downarrow \text{ の場合} \\ \uparrow & \text{それ以外の場合} \end{cases}$$

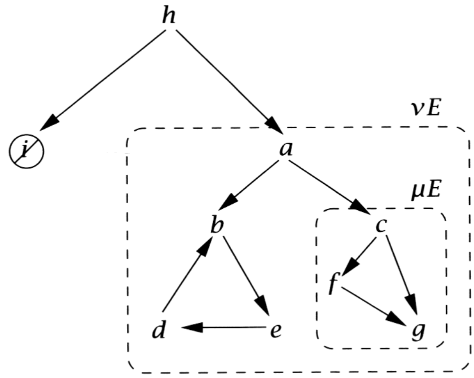
文脈から F が明らかである場合には、$support_F$ の添字を省略することが多い (後で定義する F に基づいた似た関数についても同様である)。 □

演習 21.5.2. [★★] 定義 21.3.1 と定義 21.3.2 での、部分型関係の生成関数 S_f と S が、それぞれ可逆であることを検証せよ。また、それぞれの支持関数を示せ。 □

我々の目的は、生成関数 F の最小不動点および最大不動点に対する所属性検査を行うアルゴリズムの開発である。これらアルゴリズムの基本的な手順は「F の逆向きの実行」を用いるというものである。すなわち、要素 x の所属性を検査するために、x が F によってどのように生成されうるかを問う必要がある。F が可逆であることの利点は、与えられた x を生成する方法が高々一つしかないことである。可逆でない F の場合、要素は複数の方法で生成されうるため、アルゴリズムが探索しなくてはならない経路の数の組み合わせ爆発を招く。以降では、可逆な生成関数にのみ注目する。

定義 21.5.3. 要素 x は $support_F(x)\downarrow$ であるときに「F に支持されている」といい、そうでないと

[†2] $support$ のような関数 f は、特別な値「失敗」を値域に含むとする。記法 $f(x)\downarrow$ は、f を x へと適用した結果が値になることを表し、$f(x)\uparrow$ は f が x で失敗することを表す。後者を $f(x) = \uparrow$ と表すこともある。



▲ 図 21-2. 支持関数の例

きに「F に支持されていない」という。F に支持されている要素は $support_F(x) = \emptyset$ であるときに「F の基底である」という。 □

支持されていない要素 x はどんな X をとっても $F(X)$ には現れない。一方、基底である x は任意の X について $F(X)$ の要素になっている。

可逆な関数は、**支持グラフ**として可視化できる。例えば、図 21-2 は、普遍集合 $\{a, b, c, d, e, f, g, h, i\}$ 上の関数 E を、普遍集合のある要素を支持するために、どんな要素が必要となるか示すことによって定義している。すなわち、与えられた要素 x に対して、$support_E(x)$ は x から矢印の伸びている任意の y を含む。支持されていない要素は斜線入りの丸で表している。この例では、i は唯一の支持されていない要素であり、g は唯一の基底である要素である。(我々の定義によれば、h は、その支持集合として支持されていない要素を含むものの、**支持されている**ことに注意されたい。)

演習 21.5.4. [★] この関数に対応する推論規則を、例 21.1.3 にならって与えよ。$E(\{b, c\}) = \{g, a, d\}$ であること、$E(\{a, i\}) = \{g, h\}$ であること、μE と νE とマークされた要素の集合が実際に E の最小不動点および最大不動点になっていることを確認せよ。 □

図 21-2 のグラフを考えると、要素 x が最大不動点に含まれるのは、支持グラフ上で x からどの非支持要素にも到達できないとき、かつそのときに限るというアイディアが浮かぶ。ここから、x が νF の要素であることを検査するのに、支持関数を通じて x から到達可能な要素をすべて列挙するという、アルゴリズムの戦略が示唆される。列挙結果に非支持要素が現れた場合には失敗を返し、そうでない場合には成功とする。ただし、要素間の到達可能性には循環が存在しうるので、列挙手続きが無限ループに陥らないよう予防策を講じなくてはならないことに注意されたい。この節では以降、このアイディアを追求する。

定義 21.5.5. F を可逆な生成関数とする。ブール値関数 gfp_F (単に gfp とも書く)[†3]を、以下のように定義する[†4]。

$$
\begin{aligned}
gfp(X) \quad = \quad & \text{if } support(X)\uparrow \text{ then } false \\
& \text{else if } support(X) \subseteq X \text{ then } true \\
& \text{else } gfp(support(X) \cup X)
\end{aligned}
$$

---

†3 (訳注) gfp は greatest fixed point の略。

†4 再帰関数を定義するための標準的な記法をここでは用いている。すなわち、gfp は等式を満たす**最小の**部分関数を意図している。このような定義はより形式的には、それ自体、適切な生成関数の最小不動点として見ることができる。詳細については、例えば Gunter [188]、Winskel [496]、Mitchell [322] の教科書などに載っている、表示的意味論の標準的な取り扱いを参照せよ。

直観的には、gfp は X から始め、F について整合的になるか支持されていない要素が見つかるかするまで、支持関数を用いて拡大していく。$gfp(x) = gfp(\{x\})$ と定義することで、gfp の定義を個々の要素へも拡張する。 □

演習 21.5.6. [★]　図 21-2 から得られる別の観察結果として、νF の要素 x は、x が支持グラフでの循環に含まれる場合（もしくは、x から循環に含まれる要素へのパスが存在する場合）には、μF の要素にはならないという観察がある。この逆もまた正しいだろうか。すなわち、x が νF の要素であって μF の要素でないとき、必ず x から循環へと至ることができるだろうか。 □

本節のこれ以降の部分を、gfp の正当性と停止性の証明にあてる（初読の際にはこれを飛ばして次の節に行ってしまってもよい）。まず、支持関数の性質を二つ確認するところから始める。

補題 21.5.7. $X \subseteq F(Y)$ であるのは、$support_F(X)\downarrow$ かつ $support_F(X) \subseteq Y$ であるとき、かつそのときに限る。 □

[証明]　$x \in F(Y)$ であるのが、$support(x)\downarrow$ かつ $support(x) \subseteq Y$ であるとき、かつそのときに限ることを示せば十分である。まず、$x \in F(Y)$ と仮定する。すると、$Y \in G_x = \{X \subseteq \mathcal{U} \mid x \in F(X)\}$、すなわち $G_x \neq \emptyset$ である。したがって、F が可逆であることから、G_x に含まれる最小の集合 $support(x)$ が存在し、$support(x) \subseteq Y$ である。逆に、$support(x) \subseteq Y$ とすると、単調性より $F(support(x)) \subseteq F(Y)$ がいえる。しかし、$support$ の定義より、$x \in F(support(x))$ であるため、$x \in F(Y)$ である。 □

補題 21.5.8. P を F の不動点とする。このとき、$X \subseteq P$ であるのは $support_F(X)\downarrow$ かつ $support_F(X) \subseteq P$ であるとき、かつそのときに限られる。 □

[証明]　$P = F(P)$ であることを思い出し、補題 21.5.7 を適用すればよい。 □

これで gfp の部分正当性を証明できる。（生成関数によっては gfp は発散するため、ここでは全正当性についてはまだ扱わない。この節の後のほうで生成関数の制限されたクラスに対しての停止性を証明する。）

定理 21.5.9.

(1)　$gfp_F(X) = true$ ならば $X \subseteq \nu F$
(2)　$gfp_F(X) = false$ ならば $X \nsubseteq \nu F$ □

[証明]　各節の証明は、アルゴリズムの実行の再帰的構造に関する帰納法によって行う。

(1)　gfp の定義から、$gfp(X)$ が $true$ を返すことのできる場合が二つであることがわかる。もし、$support(X) \subseteq X$ であるために $gfp(X) = true$ である場合には、補題 21.5.7 により、$X \subseteq F(X)$ すなわち X が F について整合的であることがわかる。したがって、余帰納法原理により $X \subseteq \nu F$ である。一方、$gfp(support(X) \cup X) = true$ であるために $gfp(X) = true$ である場合には、帰納法の仮定から、$support(X) \cup X \subseteq \nu F$ であり、したがって $X \subseteq \nu F$ である。

(2)　$gfp(X) = false$ になる場合も二通りある。まず、$support(X)\uparrow$ であるため $gfp(X) = false$ となる場合を考えよう。このときには、補題 21.5.8 により $X \nsubseteq \nu F$ である。次に、$gfp(support(X) \cup X) = false$ であるために $gfp(X) = false$ である場合を考えよう。このとき帰納法の仮定より $support(X) \cup X \nsubseteq \nu F$ である。これは、言い換えれば $X \nsubseteq \nu F$ もしくは

$support(X) \notin \nu F$ であるということである。どちらの場合にも $X \notin \nu F$ である（後者の場合には補題 21.5.8 を用いる）。 □

次に、*gfp* の停止性の十分条件を同定し、このアルゴリズムが停止することが保証される生成関数のクラスを与える。このクラスを記述するために、幾つか新たな用語が必要になる。

定義 21.5.10. 可逆な生成関数 F と要素 $x \in \mathcal{U}$ が与えられたとき、x の直接の前者（predecessor）の集合 $pred_F(x)$（もしくは単に $pred(x)$）を以下のように定義する。

$$pred(x) = \begin{cases} \emptyset & support(x)\uparrow \text{ の場合} \\ support(x) & support(x)\downarrow \text{ の場合} \end{cases}$$

また、集合 $X \subseteq \mathcal{U}$ に対しても以下のように拡張する。

$$pred(X) = \bigcup_{x \in X} pred(x)$$

X から支持関数を用いて到達可能（reachable）なすべての要素からなる集合 $reachable_F(X)$（あるいは単に $reachable(X)$）を以下のように定義する。

$$reachable(X) = \bigcup_{n \geq 0} pred^n(X)$$

また、単一の要素 $x \in \mathcal{U}$ への拡張は以下のように定義する。

$$reachable(x) = reachable(\{x\})$$

$y \in reachable(x)$ であるときに、要素 $y \in \mathcal{U}$ は要素 x から**到達可能**であるという。 □

定義 21.5.11. 可逆な生成関数 F は、各 $x \in \mathcal{U}$ について $reachable(x)$ が有限であるとき、**有限状態**であるという。 □

有限状態の生成関数では、*gfp* の探索空間は有限であるので、*gfp* は必ず停止する。

定理 21.5.12. $reachable_F(X)$ が有限であれば、$gfp_F(X)$ は定義される。したがって、F が有限状態であれば、$gfp_F(X)$ は任意の有限の $X \subseteq \mathcal{U}$ に対して停止する。 □

［証明］ $gfp(X)$ の元の呼び出しから生成される呼び出しグラフ中の、$gfp(Y)$ の各再帰呼び出しについて、$Y \subseteq reachable(X)$ が成り立つ。さらに、Y は各呼び出しで真に増大する。$reachable(X)$ は有限なので、$m(Y) = |reachable(X)| - |Y|$ は *gfp* の停止尺度の役割を果たす。 □

演習 21.5.13. ［★★★］ F を可逆な生成関数とする。関数 lfp_F（あるいは単に lfp）を以下のように定義せよ。

$$\begin{aligned} lfp(X) \quad = \quad & \text{if } support(X)\uparrow \text{ then } false \\ & \text{else if } X = \emptyset \text{ then } true \\ & \text{else } lfp(support(X)) \end{aligned}$$

直観的には、*lfp* は集合 X から始めて、支持関係を使ってこれを空になるまで縮小していく。このアルゴリズムの以下の意味での部分正当性を証明せよ。

(1) $lfp_F(X) = true$ ならば $X \subseteq \mu F$
(2) $lfp_F(X) = false$ ならば $X \notin \mu F$

生成関数のクラスで、lfp_F がそのすべての有限の入力に対して停止するようなものを見つけられるだろうか。 □

21.6 より効率的なアルゴリズム

gfp のアルゴリズムは正しいが、再帰呼び出しを行うたびに毎回集合 X 全体の支持集合を再計算しているため、あまり効率はよくない。例えば、以下は図 21-2 の関数 E に対して *gfp* を適用した際のトレースである。

$$\begin{array}{ll} & gfp(\{a\}) \\ = & gfp(\{a,b,c\}) \\ = & gfp(\{a,b,c,e,f,g\}) \\ = & gfp(\{a,b,c,e,f,g,d\}) \\ = & true \end{array}$$

ここで、$support(a)$ が四回計算されていることに注意されたい。この余分な再計算は、アルゴリズムの改良で除去できる。*support* が既に考慮済みであるような**仮定**の集合 A と、*support* をまだ考慮していない**ゴール**の集合 X を保持すればよい。

定義 21.6.1. F を可逆な生成関数とする。関数 gfp_F^a（あるいは単に gfp^a）を以下のように定義する（添字 “a” は仮定（assumption）を意味する）。

$$\begin{array}{lcl} gfp^a(A,X) & = & \text{if } support(X)\uparrow \text{ then } false \\ & & \text{else if } X = \varnothing \text{ then } true \\ & & \text{else } gfp^a(A \cup X,\ support(X) \setminus (A \cup X)) \end{array}$$

$x \in \nu F$ を検査するには、$gfp^a(\varnothing, \{x\})$ を計算すればよい。 □

このアルゴリズム（とこの節の後で扱う二つのアルゴリズム）は、各要素の支持集合を高々一度しか計算しない。上と同じ例に対するトレースは以下のようになる。

$$\begin{array}{ll} & gfp^a(\varnothing,\ \{a\}) \\ = & gfp^a(\{a\},\ \{b,c\}) \\ = & gfp^a(\{a,b,c\},\ \{e,f,g\}) \\ = & gfp^a(\{a,b,c,e,f,g\},\ \{d\}) \\ = & gfp^a(\{a,b,c,e,f,g,d\},\ \varnothing) \\ = & true \end{array}$$

必然的に、このアルゴリズムの正当性の言明は、前節でのものに比べ、若干精緻なものになる。

定理 21.6.2.

(1) $support_F(A) \subseteq A \cup X$ かつ $gfp_F^a(A,X) = true$ ならば、$A \cup X \subseteq \nu F$
(2) $gfp_F^a(A,X) = false$ ならば、$X \nsubseteq \nu F$ □

［証明］ 定理 21.5.9 と同様。 □

この節では以降、*gfp* アルゴリズムのもう二つの変種で、再帰型についてのよく知られた部分型付けアルゴリズムに、より密に対応するものを見ていく。初読の際には飛ばして次の節のはじめに

進んでもよい。

定義 21.6.3. gfp^a のあるちょっとした変種は、一度に X の要素を一つだけ取り出し、その支持集合を展開する。この新しいアルゴリズムのことを gfp_F^s（あるいは単に gfp^s。"s" は "single" より）と呼ぶ。

$$
\begin{aligned}
gfp^s(A,X) \quad = \quad & \text{if } X = \varnothing \text{ then } \mathit{true} \\
& \text{else let } x \text{ be some element of } X \text{ in} \\
& \quad \text{if } x \in A \text{ then } gfp^s(A, X \setminus \{x\}) \\
& \quad \text{else if } \mathit{support}(x)\uparrow \text{ then } \mathit{false} \\
& \quad \text{else } gfp^s(A \cup \{x\}, (X \cup \mathit{support}(x)) \setminus (A \cup \{x\}))
\end{aligned}
$$

このアルゴリズムの正当性の言明（すなわち再帰的な「ループ」の不変条件）は、定理 21.6.2 と全く同じである。

再帰的な部分型付けに対する既存のアルゴリズムの多くは、上のアルゴリズムと違い、引数として集合ではなく候補となる要素をただ一つだけ取る。我々のアルゴリズムをさらに少し修正することで、これらにより近いものになる。この変更をしたアルゴリズムはもはや末尾再帰的ではない[†5]。なぜならば、このアルゴリズムはまだ検査していない部分ゴールを記憶しておくのに呼び出しスタックを用いているためである。他の変更点として、このアルゴリズムは引数として仮定の集合 A を取ると同時に、返り値も新たな仮定の集合であるという点がある。これにより、処理を終えた再帰呼び出しの内部で生成された部分型付けの仮定を記録し、後の呼び出しで再利用できるようになる。事実上、仮定の集合が再帰呼び出しのグラフを通じて「直列的に受け渡し（threading）」されることになり、そのためこのアルゴリズムを gfp^t と呼ぶ。

定義 21.6.4. 可逆な生成関数 F が与えられたとき、gfp_F^t（あるいは単に gfp^t）を以下のように定義する。

$$
\begin{aligned}
gfp^t(A,x) \quad = \quad & \text{if } x \in A \text{ then } A \\
& \text{else if } \mathit{support}(x)\uparrow \text{ then } \mathit{fail} \\
& \text{else} \\
& \quad \text{let } \{x_1, \ldots, x_n\} = \mathit{support}(x) \text{ in} \\
& \quad \text{let } A_0 = A \cup \{x\} \text{ in} \\
& \quad \text{let } A_1 = gfp^t(A_0, x_1) \text{ in} \\
& \quad \ldots \\
& \quad \text{let } A_n = gfp^t(A_{n-1}, x_n) \text{ in} \\
& \quad A_n
\end{aligned}
$$

$x \in \nu F$ であることを検査するには $gfp^t(\varnothing, x)$ を計算すればよい。もしこの呼び出しが成功すれば $x \in \nu F$ であり、失敗すれば $x \notin \nu F$ である。失敗の扱いに関して、B が失敗する際には「let $A = B$ in C」も失敗するという規約を用いる。これにより、gfp^t の再帰呼び出しごとに明示的な「例外処理」節を書かずに済む。 □

このアルゴリズムの正当性の言明は、上で用いたものから再度改良し、非末尾再帰的な性質を考

[†5] **末尾再帰**呼び出し（もしくは**末尾呼び出し**）は、呼び出し元の関数の最後の動作であるような再帰呼び出しである。すなわち、その再帰呼び出しから返された結果が、呼び出し元の結果にもなる再帰呼び出しである。末尾呼び出しについて興味深いのは、関数型言語のコンパイラの多くが、末尾呼び出しを単純な分岐命令として実装することである。このとき再帰呼び出しのための新たなスタックフレームは確保されず、呼び出し側のスタックが再利用される。そのため、末尾再帰関数として実装されたループは、等価な `while` ループと同じ機械語へとコンパイルされるのである。

慮に入れるため、支持集合を検査する必要のある要素の「スタック」 X を追加で配置しなくてはならない。

補題 21.6.5.

(1) $gfp^t_F(A, x) = A'$ ならば $A \cup \{x\} \subseteq A'$ である。
(2) 任意の X について、もし $support_F(A) \subseteq A \cup X \cup \{x\}$ かつ $gfp^t_F(A, x) = A'$ ならば、$support_F(A') \subseteq A' \cup X$ である。 □

[証明] (1) はこのアルゴリズムの実行の再帰的構造に関する定型的な帰納法で証明できる。

(2) もこのアルゴリズムの実行の再帰的な構造に関する帰納法で証明する。もし、$x \in A$ ならば、$A' = A$ であり、欲している結論は仮定から即座にいえる。一方、$A' \neq A$ であったとし、さらに特別な場合として $support(x)$ が二つの要素 x_1, x_2 を含んでいる場合を考えよう（二つに限らない一般の場合の証明は、ここでは示さないが、$support(x)$ のサイズに関する帰納法を内側で使うことで証明できる）。このアルゴリズムは、A_0, A_1, A_2 を計算し、A_2 を返す。任意の X_0 について、もし $support(A) \subseteq A \cup \{x\} \cup X_0$ ならば、$support(A_2) \subseteq A_2 \cup X_0$ であることを示したい。まず、$X_1 = X_0 \cup \{x_2\}$ とする。このとき

$$
\begin{aligned}
support(A_0) &= support(A) \cup support(x) \\
&= support(A) \cup \{x_1, x_2\} \\
&\subseteq A \cup \{x\} \cup X_0 \cup \{x_1, x_2\} \\
&= A_0 \cup X_0 \cup \{x_1, x_2\} \\
&= A_0 \cup X_1 \cup \{x_1\}
\end{aligned}
$$

が成り立つので、帰納法の仮定における全称量化された X を X_1 へと具体化し、最初の再帰呼び出しへ適用できる。それにより $support(A_1) \subseteq A_1 \cup X_1 = A_1 \cup \{x_2\} \cup X_0$ が得られる。次に、帰納法の仮定における全称量化された X を X_0 へと具体化して、二つ目の再帰呼び出しへと適用することで、望む結果 $support(A_2) \subseteq A_2 \cup X_0$ が得られる。 □

定理 21.6.6.

(1) $gfp^t_F(\emptyset, x) = A'$ ならば $x \in \nu F$
(2) $gfp^t_F(\emptyset, x) = fail$ ならば $x \notin \nu F$ □

[証明] (1) に関しては、補題 21.6.5 (1) により、$x \in A'$ であることに着目せよ。同補題の (2) を $X = \emptyset$ で具体化すると $support(A') \subseteq A'$ が得られる。すなわち、補題 21.5.7 により、A' は F について整合的であり、ここから余帰納法により、$A' \subseteq \nu F$ である。(2) に関しては、ある A が存在して、$gfp^t_F(A, x) = fail$ ならば、$x \notin \nu F$ である、ということを、（補題 21.5.8 を用いた、gfp^t_F アルゴリズムの実行の深さに関する簡単な帰納法を用いて）主張する。 □

この節のアルゴリズムはすべて到達可能な要素の集合しか確かめないため、停止する十分条件は元の gfp アルゴリズムと同じである。すなわち、F が有限状態であれば、すべての入力に対して停止する。

21.7 正則木

これまでに、生成関数 F の最大不動点として定義される集合への所属性を検査する一般的なアルゴリズムを複数、F が可逆かつ有限状態という仮定の下で開発してきた。また、無限木の間の部分型関係をある種の生成関数 S の最大不動点として定義する方法を示してきた。次のステップは当然、これらのアルゴリズムのいずれかを S に対して具体化することである。もちろん、この具体化されたアルゴリズムはすべての入力に対して停止するわけではない。これは、無限型の二つ組から到達可能な状態は一般には無限である可能性があるためである。しかし、この節で見るように、ある種のよい振る舞いをする形の無限型である、**正則型**に対象を絞ることで、到達可能な状態が有限にとどまることが保証でき、部分型検査アルゴリズムが常に停止するようになる。

定義 21.7.1. 木型 S が木型 T の**部分木**であるとは、$\mathsf{S} = \lambda\sigma.\,\mathsf{T}(\pi,\sigma)$ となる π が存在することをいう。すなわち、パスから記号への関数 S が、関数 T に与えるパスの引数に定数 π を前置することで、関数 T から得られる場合である。接頭辞 π は、T の根から S の根へのパスに対応する。T のすべての部分木の集合を *subtrees*(T) で表す。 □

定義 21.7.2. 木型 $\mathsf{T} \in \mathcal{T}$ は、*subtrees*(T) が有限であるとき、すなわち異なった部分木を有限個しか持たないときに、**正則**であるという。正則な木型の集合を $\mathcal{T}_r$ で表す。 □

例 21.7.3.

(1) 有限木型は正則である（異なる部分木の数は高々ノードの数）。異なる部分木の数はノード数よりも真に小さい場合がある。例えば、T = Top→(Top × Top) は五つのノードを持つが、異なる部分木は三つしかない（T それ自体、Top × Top、Top）。

(2) 無限木の中にも正則なものが存在する。例えば、

T = Top × (Top × (Top × ...))

という木は異なる部分木を二つ（T 自身と Top）しか持たない。

(3) 木型

T = B × (A × (B × (A × (A × (B × (A × (A × (A × (B × ...)

において、隣接する B 同士の間を区切る A の個数が増えていくとすると、この木型は正則ではない。T は正則ではないので、T<:T という部分型付け判断式を正当化するために必要な部分型付けの組すべてを含む集合 $\mathit{reachable}_{S}(\mathsf{T},\mathsf{T})$ は、無限集合となる。 □

命題 21.7.4. 生成関数 S の正則木型への制限 S_r は、有限状態である。 □

［証明］ 正則木型の任意の二つ組 (S, T) に対して、集合 $\mathit{reachable}_{S_r}(\mathsf{S},\mathsf{T})$ が有限であることを示す必要がある。$\mathit{reachable}_{S_r}(\mathsf{S},\mathsf{T}) \subseteq (\mathit{subtrees}(\mathsf{S}) \times \mathit{subtrees}(\mathsf{T})) \cup (\mathit{subtrees}(\mathsf{T}) \times \mathit{subtrees}(\mathsf{S}))$ であることに着目すると、*subtrees*(S) と *subtrees*(T) は共に有限なので、右辺は有限である。 □

これが意味するのは、いずれかの所属性検査アルゴリズムを S で具体化することで、正則木型の部分型関係の決定手続きが得られるということである。当然ながら、これが実際の実装でうまく機能するためには、正則木が有限の構造で表現されていなければならない。そのような表現方法として、次節で取り上げる μ 記法がある。

21.8 μ 型

この節では有限の μ 記法を説明し、μ 式上の部分型関係を定義し、この部分型関係と木型の部分型関係を対応させる。

定義 21.8.1. X は型変数からなるある固定された可算集合 $\{X_1, X_2, \ldots\}$ の上を動くとする。**生** (raw) **の** μ **型**の集合 $\mathcal{T}_m^{raw}$ を、以下の文法で定義される式の集合とする。

$$
\begin{array}{lll}
\mathsf{T} & ::= & \mathsf{X} \\
& & \mathsf{Top} \\
& & \mathsf{T}\times\mathsf{T} \\
& & \mathsf{T}\to\mathsf{T} \\
& & \mu\mathsf{X}.\mathsf{T}
\end{array}
$$

構文記号 μ は束縛子であり、束縛変数と自由変数、閉じた生の μ 型、生の μ 型の束縛変数名の付け替えを同一視した同値性といった概念が、標準的な方法で生じる。$FV(\mathsf{T})$ は生の μ 型 T の自由変数の集合を表す。生の μ 型 T 中の自由な X の出現を S で置き換える捕獲回避代入 $[\mathsf{X}\mapsto\mathsf{S}]\mathsf{T}$ も通常の方法で定義する。 □

正則木との対応を緊密なものにするために、生の μ 型にもう少し制約を加えなくてはならない。木型を、与えられた μ 型を無限に展開したものとして「読み上げ」られるようにしたいわけだが、生の μ 型には、木型の表現として妥当な解釈ができないものがあるのである。それらの型は $\mu\mathsf{X}.\mu\mathsf{X}_1\ldots\mu\mathsf{X}_n.\mathsf{X}$ という形の部分式を持つ。ここで $\mathsf{X}_1 \ldots \mathsf{X}_n$ は X とは異なる変数である。例えば、$\mathsf{T} = \mu\mathsf{X}.\mathsf{X}$ という例を考えよう。T を展開すると再度 T になるため、T を展開してもどんな木も読み上げられない。これにより以下の制限へと行きつく。

定義 21.8.2. 生の μ 型 T は、T の $\mu\mathsf{X}.\mu\mathsf{X}_1\ldots\mu\mathsf{X}_n.\mathsf{S}$ という形の任意の部分式について、本体 S が X ではないときに、**縮小的**であるという。これは、μ で束縛された変数の本体の中でのすべての出現が、少なくとも一つの → もしくは × によって束縛子から分離されているときに生の μ 型は縮小的である、としても同値である。

生の μ 型は縮小的であるときに単に μ **型**と呼ぶ。μ 型の集合を $\mathcal{T}_m$ と書く。

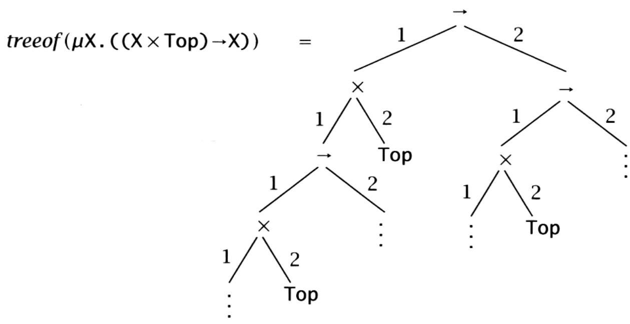
T が μ 型であるとき、T の先頭にある μ 束縛の数を $\mu\text{-}height(T)$ と書く。 □

μ 型が、無限の正則木型の、有限の表記法であるという共通認識は、以下の関数によって形式化される。

定義 21.8.3. 閉じた μ 型を木型に写す関数 *treeof* を以下のように帰納的に定義する。

$$
\begin{array}{lll}
treeof(\mathsf{Top})(\bullet) & = & \mathsf{Top} \\[1ex]
treeof(\mathsf{T}_1\to\mathsf{T}_2)(\bullet) & = & \to \\
treeof(\mathsf{T}_1\to\mathsf{T}_2)(i,\pi) & = & treeof(\mathsf{T}_i)(\pi) \\[1ex]
treeof(\mathsf{T}_1\times\mathsf{T}_2)(\bullet) & = & \times \\
treeof(\mathsf{T}_1\times\mathsf{T}_2)(i,\pi) & = & treeof(\mathsf{T}_i)(\pi) \\[1ex]
treeof(\mu\mathsf{X}.\mathsf{T})(\pi) & = & treeof([\mathsf{X}\mapsto\mu\mathsf{X}.\mathsf{T}]\mathsf{T})(\pi)
\end{array}
$$

この定義が適切なものである（すなわち網羅的でかつ停止する）ことを確認するには、以下に着目せよ。



▲ 図 21-3.　*treeof* の適用例

(1) 右辺での *treeof* の再帰的な使用は、二つ組 $(|\pi|, \mu\text{-}height(\mathsf{T}))$ の辞書式でのサイズを減少させる。$\mathsf{S}\to\mathsf{T}$ と $\mathsf{S}\times\mathsf{T}$ の場合には $|\pi|$ が減少し、$\mu\mathsf{X}.\mathsf{T}$ の場合には $|\pi|$ はそのままだが $\mu\text{-}height(\mathsf{T})$ が減少する。
(2) すべての再帰呼び出しは、引数の型が縮小的であることと、閉じていることを保つ。特に、型 $\mu\mathsf{X}.\mathsf{T}$ が縮小的かつ閉じているのは、その展開 $[\mathsf{X}\mapsto\mu\mathsf{X}.\mathsf{T}]\mathsf{T}$ がそうであるとき、かつそのときに限られる。これにより $treeof(\mu\mathsf{X}.\mathsf{T})$ の定義中の展開ステップが正当化される。

型の二つ組についても、*treeof* 関数を $treeof(\mathsf{S},\mathsf{T}) = (treeof(\mathsf{S}), treeof(\mathsf{T}))$ と定義して持ち上げておく。 □

μ 型への *treeof* の適用例を図 21-3 に示す。

21.3 節で木型の部分型関係を、生成関数 S の最大不動点として定義した。この節では型の構文を μ 型によって拡張したが、その振る舞いは**右 μ 畳み込み**と**左 μ 畳み込み**という二つの **μ 畳み込み**規則によって直観的に記述できる。

$$\frac{\mathsf{S} <: [\mathsf{X}\mapsto\mu\mathsf{X}.\mathsf{T}]\mathsf{T}}{\mathsf{S} <: \mu\mathsf{X}.\mathsf{T}} \quad \text{および} \quad \frac{[\mathsf{X}\mapsto\mu\mathsf{X}.\mathsf{S}]\mathsf{S} <: \mathsf{T}}{\mu\mathsf{X}.\mathsf{S} <: \mathsf{T}}$$

形式的には、S の定義と同じ三つの節と、μ 畳み込みに対応する二つの追加的な節からなる、生成関数 S_m を与えることによって、μ 型の部分型関係を定義する。

定義 21.8.4. 二つの μ 型 S と T は $(\mathsf{S},\mathsf{T}) \in \nu S_m$ であるときに、部分型関係にあるという。ここで $S_m \in \mathcal{P}(\mathcal{T}_m \times \mathcal{T}_m) \to \mathcal{P}(\mathcal{T}_m \times \mathcal{T}_m)$ は以下で定義される単調関数である。

$$\begin{array}{rcl} S_m(R) & = & \{(\mathsf{S},\mathsf{Top}) \mid \mathsf{S} \in \mathcal{T}_m\} \\ & \cup & \{(\mathsf{S}_1\times\mathsf{S}_2, \mathsf{T}_1\times\mathsf{T}_2) \mid (\mathsf{S}_1,\mathsf{T}_1), (\mathsf{S}_2,\mathsf{T}_2) \in R\} \\ & \cup & \{(\mathsf{S}_1\to\mathsf{S}_2, \mathsf{T}_1\to\mathsf{T}_2) \mid (\mathsf{T}_1,\mathsf{S}_1), (\mathsf{S}_2,\mathsf{T}_2) \in R\} \\ & \cup & \{(\mathsf{S}, \mu\mathsf{X}.\mathsf{T}) \mid (\mathsf{S}, [\mathsf{X}\mapsto\mu\mathsf{X}.\mathsf{T}]\mathsf{T}) \in R\} \\ & \cup & \{(\mu\mathsf{X}.\mathsf{S}, \mathsf{T}) \mid ([\mathsf{X}\mapsto\mu\mathsf{X}.\mathsf{S}]\mathsf{S}, \mathsf{T}) \in R,\ \text{かつ}\ \mathsf{T} \neq \mathsf{Top},\ \text{かつ}\ \mathsf{T} \neq \mu\mathsf{Y}.\mathsf{T}_1\} \end{array}$$

この定義は上の μ 畳み込み規則を正確に具現化したものではないことに注意されたい。最後の節とその前の節の間に非対称性を導入し可逆にしている（そうしないと節がオーバーラップしてしまう）。しかし、次の演習でわかるように、S_m はより自然な生成関数[†6] S_d と同じ部分型関係を生成する。S_d の節は推論規則と正確に対応している。 □

[†6] S_d 中の “d” は、μ 畳み込みの、S_m で用いている「アルゴリズム的」な推論規則ではなく、「宣言的（declarative）」な推論規則に基づいた関数であることを示している。

演習 21.8.5. [★★★] 上で述べた関数 S_d を書き下し、可逆ではないことを実例によって示せ。また、$\nu S_d = \nu S_m$ であることを証明せよ。 □

対応する支持関数が正しく定義できるので、この生成関数 S_m は可逆である。

$$support_{S_m}(\mathsf{S},\mathsf{T}) = \begin{cases} \emptyset & \mathsf{T} = \mathsf{Top} \text{ の場合} \\ \{(\mathsf{S}_1,\mathsf{T}_1), (\mathsf{S}_2,\mathsf{T}_2)\} & \mathsf{S} = \mathsf{S}_1 \times \mathsf{S}_2 \text{ かつ} \\ & \mathsf{T} = \mathsf{T}_1 \times \mathsf{T}_2 \text{ の場合} \\ \{(\mathsf{T}_1,\mathsf{S}_1), (\mathsf{S}_2,\mathsf{T}_2)\} & \mathsf{S} = \mathsf{S}_1 \to \mathsf{S}_2 \text{ かつ} \\ & \mathsf{T} = \mathsf{T}_1 \to \mathsf{T}_2 \text{ の場合} \\ \{(\mathsf{S}, [\mathsf{X} \mapsto \mu\mathsf{X}.\mathsf{T}_1]\mathsf{T}_1)\} & \mathsf{T} = \mu\mathsf{X}.\mathsf{T}_1 \text{ の場合} \\ \{([\mathsf{X} \mapsto \mu\mathsf{X}.\mathsf{S}_1]\mathsf{S}_1, \mathsf{T})\} & \mathsf{S} = \mu\mathsf{X}.\mathsf{S}_1 \text{ かつ} \\ & \mathsf{T} \neq \mu\mathsf{X}.\mathsf{T}_1, \mathsf{T} \neq \mathsf{Top} \text{ の場合} \\ \uparrow & \text{それ以外の場合} \end{cases}$$

ここまで μ 型上の部分型関係を、以前に定義した木型上の部分型関係とは独立して導入してきた。μ 型を正則型の有限の形での表現方法として考えているので、二つの部分型関係の概念がお互いに対応していることを保証しなくてはならない。次の定理 21.8.7 はこの対応を確立する。しかしその前に技術的な補題を一つ証明しておく必要がある。

補題 21.8.6. $R \subseteq \mathcal{T}_m \times \mathcal{T}_m$ が S_m について整合的であるとする。任意の $(\mathsf{S},\mathsf{T}) \in R$ について、ある $(\mathsf{S}',\mathsf{T}') \in R$ が存在して、$treeof(\mathsf{S}',\mathsf{T}') = treeof(\mathsf{S},\mathsf{T})$ を満たし、かつ S' も T' も μ で始まらない。 □

[証明] S と T の先頭にある μ の総数に関する帰納法による。もし、S も T も μ で始まらなければ、$(\mathsf{S}',\mathsf{T}') = (\mathsf{S},\mathsf{T})$ とすればよい。一方、もし $(\mathsf{S},\mathsf{T}) = (\mathsf{S},\mu\mathsf{X}.\mathsf{T}_1)$ ならば、R の S_m についての整合性により、$(\mathsf{S},\mathsf{T}) \in S_m(R)$ がいえ、$(\mathsf{S}'',\mathsf{T}'') = (\mathsf{S},[\mathsf{X} \mapsto \mu\mathsf{X}.\mathsf{T}_1]\mathsf{T}_1) \in R$ となる。T は縮小的であるので、T の展開結果 T'' が先頭に持つ μ の数は T よりも 1 小さくなっている。帰納法の仮定から、ある $(\mathsf{S}',\mathsf{T}') \in R$ が存在し、S' も T' も μ で始まっておらず、かつ $treeof(\mathsf{S}'',\mathsf{T}'') = treeof(\mathsf{S}',\mathsf{T}')$ である。$treeof$ の定義により、$treeof(\mathsf{S},\mathsf{T}) = treeof(\mathsf{S}'',\mathsf{T}'')$ であるので、二つ組 $(\mathsf{S}',\mathsf{T}')$ が必要としていたものである。$(\mathsf{S},\mathsf{T}) = (\mu\mathsf{X}.\mathsf{S}_1,\mathsf{T})$ の場合についても同様である。 □

定理 21.8.7. $(\mathsf{S},\mathsf{T}) \in \mathcal{T}_m \times \mathcal{T}_m$ とする。このとき、$treeof(\mathsf{S},\mathsf{T}) \in \nu S$ は $(\mathsf{S},\mathsf{T}) \in \nu S_m$ の必要十分条件である。 □

[証明] まず最初に必要条件、すなわち $(\mathsf{S},\mathsf{T}) \in \nu S_m$ ならば $treeof(\mathsf{S},\mathsf{T}) \in \nu S$ となることを考えよう。$(\mathsf{A},\mathsf{B}) = treeof(\mathsf{S},\mathsf{T}) \in \mathcal{T} \times \mathcal{T}$ とする。余帰納法の原理により、S について整合的である集合 $Q \in \mathcal{T} \times \mathcal{T}$ で $(\mathsf{A},\mathsf{B}) \in Q$ を満たすものを示せれば、この結果がいえる。そこで $Q = treeof(\nu S_m)$[†7]がそのような集合であるということを示そう。これを確かめるため、任意の $(\mathsf{A}',\mathsf{B}') \in Q$ について $(\mathsf{A}',\mathsf{B}') \in S(Q)$ であることを示さなければならない。

$(\mathsf{S}',\mathsf{T}') \in \nu S_m$ を μ 型の二つ組で $treeof(\mathsf{S}',\mathsf{T}') = (\mathsf{A}',\mathsf{B}')$ であるものとする。補題 21.8.6 より、S' も T' も μ で始まらないと仮定してよい。νS_m は S_m について整合的であるので、$(\mathsf{S}',\mathsf{T}')$ は S_m の定義のいずれかの節によって支持されていなくてはならない。すなわち、以下のいずれかの形になっていなくてはならない。

[†7] (訳注) ここでは $treeof(\nu S_m)$ は、$treeof$ を各要素に対して適用した $\{treeof(\mathsf{T}) \mid \mathsf{T} \in \nu S_m\}$ を表す。

$(S', T') = (S', \mathsf{Top})$ の場合

この場合、$B' = \mathsf{Top}$ であり、S の定義より $(A', B') \in S(Q)$ 。

$(S', T') = (S_1 \times S_2, T_1 \times T_2)$ で $(S_1, T_1), (S_2, T_2) \in \nu S_m$ の場合

treeof の定義により、$B_i = \mathit{treeof}(T_i)$ とおいて $B' = \mathit{treeof}(T') = B_1 \times B_2$ である。同様に、$A_i = \mathit{treeof}(S_i)$ とおいて $A' = A_1 \times A_2$ である。*treeof* を (S_1, T_1) および (S_2, T_2) へと適用すると $(A_1, B_1), (A_2, B_2) \in Q$ が得られる。しかし、ここで S の定義より $(A, B) = (A_1 \times A_2, B_1 \times B_2) \in S(Q)$ が得られる。

$(S', T') = (S_1 \to S_2, T_1 \to T_2)$ で $(T_1, S_1), (S_2, T_2) \in \nu S_m$ の場合

同様である。

次に、十分条件、すなわち $\mathit{treeof}(S, T) \in \nu S$ ならば $(S, T) \in \nu S_m$ を確かめよう。余帰納法の原理により、S_m について整合的である集合 $R \in \mathcal{T}_m \times \mathcal{T}_m$ で、$(S, T) \in R$ であるものを示せば十分である。$R = \{(S', T') \in \mathcal{T}_m \times \mathcal{T}_m \mid \mathit{treeof}(S', T') \in \nu S\}$ がそのような集合であることを示そう。明らかに $(S, T) \in R$ である。証明を完成させるには、$(S', T') \in R$ ならば $(S', T') \in S_m(R)$ であることを示さなくてはならない。

νS は S について整合的であるので、任意の二つ組 $(A', B') \in \nu S$ は (A', Top)、$(A_1 \times A_2, B_1 \times B_2)$、$(A_1 \to A_2, B_1 \to B_2)$ のいずれかの形でなくてはならないことに注意されたい。このことと *treeof* の定義から、任意の二つ組 $(S', T') \in R$ は (S', Top)、$(S_1 \times S_2, T_1 \times T_2)$、$(S_1 \to S_2, T_1 \to T_2)$、$(S', \mu X.T_1)$、$(\mu X.S_1, T')$ のいずれかの形でなくてはならないことがわかる。この各場合を順に考えていく。

$(S', T') = (S', \mathsf{Top})$ の場合

この場合、S_m の定義より $(S', T') \in S_m(R)$ が即座にいえる。

$(S', T') = (S_1 \times S_2, T_1 \times T_2)$ の場合

$(A', B') = \mathit{treeof}(S', T')$ とする。すると、$A_i = \mathit{treeof}(S_i)$ かつ $B_i = \mathit{treeof}(T_i)$ として $(A', B') = (A_1 \times A_2, B_1 \times B_2)$ である。$(A', B') \in \nu S$ であるので、νS の S についての整合性から、$(A_i, B_i) \in \nu S$ であることが導かれる。これはさらに R の定義より $(S_i, T_i) \in R$ を導く。S_m の定義より $(S', T') = (S_1 \times S_2, T_1 \times T_2) \in S_m(R)$ が導かれる。

$(S', T') = (S_1 \to S_2, T_1 \to T_2)$ の場合

同様である。

$(S', T') = (S', \mu X.T_1)$ の場合

$T'' = [X \mapsto \mu X.T_1]T_1$ とする。定義により $\mathit{treeof}(T'') = \mathit{treeof}(T')$ である。したがって、R の定義により $(S', T'') \in R$ がいえ、さらに S_m の定義により $(S', T') \in S_m(R)$ である。

$(S', T') = (\mu X.S_1, T')$ の場合

もし、$T' = \mathsf{Top}$ であるか、もしくは T' が μ で始まるならば、上の場合に当てはまる。そうでないならば、一つ前の場合と同様である。 □

この定理によって証明された対応は、この節で定義した μ 型間の部分型関係と、無限木型間の通常の部分型関係を有限の μ 式で表現される木型へと制限したものとの間の、健全性と完全性の言明となっている。

$$
\begin{array}{lll}
subtype(A, \mathsf{S}, \mathsf{T}) & = & \text{if } (\mathsf{S},\mathsf{T}) \in A \text{ then} \\
& & \quad A \\
& & \text{else let } A_0 = A \cup \{(\mathsf{S},\mathsf{T})\} \text{ in} \\
& & \quad \text{if } \mathsf{T} = \mathsf{Top} \text{ then} \\
& & \quad\quad A_0 \\
& & \quad \text{else if } \mathsf{S} = \mathsf{S}_1 \times \mathsf{S}_2 \text{ and } \mathsf{T} = \mathsf{T}_1 \times \mathsf{T}_2 \text{ then} \\
& & \quad\quad \text{let } A_1 = subtype(A_0, \mathsf{S}_1, \mathsf{T}_1) \text{ in} \\
& & \quad\quad subtype(A_1, \mathsf{S}_2, \mathsf{T}_2) \\
& & \quad \text{else if } \mathsf{S} = \mathsf{S}_1 \rightarrow \mathsf{S}_2 \text{ and } \mathsf{T} = \mathsf{T}_1 \rightarrow \mathsf{T}_2 \text{ then} \\
& & \quad\quad \text{let } A_1 = subtype(A_0, \mathsf{T}_1, \mathsf{S}_1) \text{ in} \\
& & \quad\quad subtype(A_1, \mathsf{S}_2, \mathsf{T}_2) \\
& & \quad \text{else if } \mathsf{T} = \mu\mathsf{X}.\mathsf{T}_1 \text{ then} \\
& & \quad\quad subtype(A_0, \mathsf{S}, [\mathsf{X} \mapsto \mu\mathsf{X}.\mathsf{T}_1]\mathsf{T}_1) \\
& & \quad \text{else if } \mathsf{S} = \mu\mathsf{X}.\mathsf{S}_1 \text{ then} \\
& & \quad\quad subtype(A_0, [\mathsf{X} \mapsto \mu\mathsf{X}.\mathsf{S}_1]\mathsf{S}_1, \mathsf{T}) \\
& & \quad \text{else} \\
& & \quad\quad fail
\end{array}
$$

▲ 図 21-4.　μ 型のための部分型付けの具体的アルゴリズム

21.9 部分式の数え上げ

汎用的なアルゴリズム gfp^t（定義 21.6.4）を、μ 型（定義 21.8.4）のための特定の支持関数 $support_{S_m}$ へと具体化することで、図 21-4 の部分型付けアルゴリズムが得られる。21.6 節の議論により、μ 型の任意の二つ組 (S, T) について $reachable_{S_m}$(S, T) が有限であれば、このアルゴリズムの停止性は保証される。この節では、これがまさに成り立つことを証明することに専念する（命題 21.9.11）。

これは一見してほとんど自明に見えるが、厳密に証明するには驚くべき労力が必要になる。困難な点は、μ 型の「閉じた部分式」の集合を定義する二通りの方法があることである。一つ目は、**トップダウン**部分式と呼ぶもので、$support_{S_m}$ で生成される部分式に直接対応する。もう一つは**ボトムアップ**部分式と呼ぶもので、任意の閉じた μ 型の部分式の集合が有限であることの単純な証明を支えている。停止性の証明は、両方の集合を定義し、前者が後者の部分集合であることを示す（命題 21.9.10）という流れになる。この節の説明は Brandt and Henglein [56] に基づいている。

定義 21.9.1. μ 型 S は μ 型 T の**トップダウン部分式**である、とは、二つ組 (S, T) が以下に示す生成関数の最小不動点に含まれることである。このことを $\mathsf{S} \sqsubseteq \mathsf{T}$ と書く。

$$
\begin{array}{lll}
TD(R) & = & \{(\mathsf{T},\mathsf{T}) \mid \mathsf{T} \in \mathcal{T}_m\} \\
& \cup & \{(\mathsf{S}, \mathsf{T}_1 \times \mathsf{T}_2) \mid (\mathsf{S}, \mathsf{T}_1) \in R\} \\
& \cup & \{(\mathsf{S}, \mathsf{T}_1 \times \mathsf{T}_2) \mid (\mathsf{S}, \mathsf{T}_2) \in R\} \\
& \cup & \{(\mathsf{S}, \mathsf{T}_1 \rightarrow \mathsf{T}_2) \mid (\mathsf{S}, \mathsf{T}_1) \in R\} \\
& \cup & \{(\mathsf{S}, \mathsf{T}_1 \rightarrow \mathsf{T}_2) \mid (\mathsf{S}, \mathsf{T}_2) \in R\} \\
& \cup & \{(\mathsf{S}, \mu\mathsf{X}.\mathsf{T}) \mid (\mathsf{S}, [\mathsf{X} \mapsto \mu\mathsf{X}.\mathsf{T}]\mathsf{T}) \in R\}
\end{array}
$$

演習 21.9.2. [★]　関係 $\mathsf{S} \sqsubseteq \mathsf{T}$ と等価な定義を、推論規則の集合によって与えよ。　□

$support_{S_m}$ の定義から、任意の μ 型 S および T について、$support_{S_m}$(S, T) に含まれるすべての二

つ組がSとTのトップダウン部分式から構成されることを容易に見て取れる。

補題 21.9.3. $(S', T') \in \mathit{support}_{S_m}(S, T)$ ならば、$S' \sqsubseteq S$ もしくは $S' \sqsubseteq T$ のどちらか一方が成り立ち、かつ、$T' \sqsubseteq S$ もしくは $T' \sqsubseteq T$ のどちらか一方が成り立つ。 □

[証明] $\mathit{support}_{S_m}$ の定義を単純に調べることで得られる。 □

また、トップダウン部分式関係は推移的である。

補題 21.9.4. $S \sqsubseteq U$ かつ $U \sqsubseteq T$ ならば $S \sqsubseteq T$ □

[証明] この補題の言明は、$\forall U, T.\ U \sqsubseteq T \Rightarrow (\forall S.\ S \sqsubseteq U \Rightarrow S \sqsubseteq T)$ に等しい。言い換えれば、$R = \{(U, T) \mid \forall S.\ S \sqsubseteq U \Rightarrow S \sqsubseteq T\}$ として、$\mu(TD) \subseteq R$ を示さなくてはならない。帰納法の原理により、R が TD について閉じている、すなわち $TD(R) \subseteq R$ であることを示せば十分である。そこで、$(U, T) \in TD(R)$ とし、TD の定義の各節の場合を見ていこう。

$(U, T) = (T, T)$ の場合
　明らかに $(T, T) \in R$ 。

$(U, T) = (U, T_1 \times T_2)$ かつ $(U, T_1) \in R$ の場合
　$(U, T_1) \in R$ であるので、すべての S について $S \sqsubseteq U \Rightarrow S \sqsubseteq T_1$ が成り立っている。また、$\sqsubseteq$ の定義により、すべての S について $S \sqsubseteq U \Rightarrow S \sqsubseteq T_1 \times T_2$ も成り立っている。したがって、R の定義により $(U, T) = (U, T_1 \times T_2) \in R$ である。

その他の場合
　同様に示せる。 □

この二つの補題を組み合わせることで、トップダウン部分式の導入の動機となる以下の命題が得られる。

命題 21.9.5. $(S', T') \in \mathit{reachable}_{S_m}(S, T)$ ならば、$S' \sqsubseteq S$ または $S' \sqsubseteq T$ であり、かつ、$T' \sqsubseteq S$ または $T' \sqsubseteq T$ である。 □

[証明] $\mathit{reachable}_{S_m}$ の定義についての帰納法で、$\sqsubseteq$ の推移律を用いればよい。 □

$\mathit{reachable}_{S_m}(S, T)$ の有限性は、上の命題と、任意の μ 型 U が有限個のトップダウン部分式しか持たないという事実から導かれる（命題 21.9.11）。残念ながら、後者の事実は $\sqsubseteq$ の定義からは明らかではない。これを TD の定義を用いて U に関する構造的帰納法によって証明しようとする試みは、TD の最後の節が帰納法の構造を破壊するので、うまくいかない。$U = \mu X.T$ の部分式を構築する際に、より**大きい**可能性のある式 $[X \mapsto \mu X.T]T$ を参照しているのである。

代替となるボトムアップ部分式の概念は、μ 型の再帰変数への代入を、部分式を計算する前にではなく**後に**行うことで、この問題を避けている。この変更により有限性を単純に証明できるようになる。

定義 21.9.6. μ 型 S が μ 型 T の**ボトムアップ部分式**であるとは、二つ組 (S, T) が以下の生成関数の最小不動点に含まれることである。このことを $S \preceq T$ と書く。

$$
\begin{array}{rcl}
BU(R) & = & \{(T, T) \mid T \in \mathcal{T}_m\} \\
 & \cup & \{(S, T_1 \times T_2) \mid (S, T_1) \in R\} \\
 & \cup & \{(S, T_1 \times T_2) \mid (S, T_2) \in R\} \\
 & \cup & \{(S, T_1 \to T_2) \mid (S, T_1) \in R\} \\
 & \cup & \{(S, T_1 \to T_2) \mid (S, T_2) \in R\} \\
 & \cup & \{([X \mapsto \mu X.T]S, \mu X.T) \mid (S, T) \in R\}
\end{array}
$$

この新たな部分式の定義が前の定義と異なるのは、μ 束縛子で始まる型の節のみである。そのような型のトップダウン部分式を得るには、まず最初に展開を行い、展開結果の部分式を集めていた。それに対して、ボトムアップ部分式を得るには、まず最初に本体の（閉じているとは限らない）部分式を集め、それから展開の代入を適用することで閉じた式へとする。

演習 21.9.7. [★★]　関係 $S \preceq T$ と等価な定義を、推論規則の集合によって与えよ。　□

式が有限個のボトムアップ部分式しか持たないという事実は容易に証明できる。

補題 21.9.8. 各 T について $\{S \mid S \preceq T\}$ は有限である。　□

[証明]　T に関する単純な構造的帰納法で、*BU* と $\preceq$ の定義から導かれる以下の観察を用いる。

- $T = Top$ もしくは $T = X$ ならば $\{S \mid S \preceq T\} = \{T\}$
- $T = T_1 \times T_2$ もしくは $T = T_1 \rightarrow T_2$ ならば $\{S \mid S \preceq T\} = \{T\} \cup \{S \mid S \preceq T_1\} \cup \{S \mid S \preceq T_2\}$
- $T = \mu X.T'$ ならば $\{S \mid S \preceq T\} = \{T\} \cup \{[X \mapsto T]S \mid S \preceq T'\}$　□

ある型のボトムアップ部分式の集合がトップダウン部分式の集合を含むことを証明するには、ボトムアップ部分式と代入を関連付ける以下の補題が必要になる。

補題 21.9.9. $S \preceq [X \mapsto Q]T$ ならば、$S \preceq Q$ が成り立つか、もしくはある $S' \preceq T$ が存在して $S = [X \mapsto Q]S'$ が成り立つ。　□

[証明]　T に関する構造的帰納法により証明する。

$T = Top$ の場合
　Top が組の右辺に現れるのは、*BU* の反射律の節のみなので、$S = Top$ でなくてはならない。$S' = Top$ をおけば所望の結果が得られる。

$T = Y$ の場合
　$Y = X$ ならば $S \preceq [X \mapsto Q]T = Q$ であり、仮定により所望の結果が成り立つ。$Y \neq X$ ならば $S \preceq [X \mapsto Q]T = Y$ である。この二つ組を正当化できるのは反射律の節のみなので、$S = Y$ でなくてはならない。$S' = Y$ とおけば所望の結果が得られる。

$T = T_1 \times T_2$ の場合
　$S \preceq [X \mapsto Q]T = [X \mapsto Q]T_1 \times [X \mapsto Q]T_2$ が成り立っている。*BU* の定義により、S がこの直積型のボトムアップ部分式になる方法が三通り存在する。この三つを順に見ていく。

　　$S = [X \mapsto Q]T$ の場合
　　　この場合には $S' = T$ とおけばよい。

　　$S \preceq [X \mapsto Q]T_1$ の場合
　　　帰納法の仮定より、$S \preceq Q$ が成り立つ（この場合にはこれで完了）か、もしくはある $S' \preceq T_1$ が存在して $S = [X \mapsto Q]S'$ が成り立つ。後者は *BU* の定義により、所望の結果 $S' \preceq T_1 \times T_2$ が導かれる。

　　$S \preceq [X \mapsto Q]T_2$ の場合
　　　同様。

$T = T_1 \rightarrow T_2$ の場合
　直積の場合と同様。

$T = \mu Y.T'$ の場合

$S \preceq [X \mapsto Q]T = \mu Y.[X \mapsto Q]T'$ が成り立っている。S がこの μ 型のボトムアップ部分式になりうる場合が二通り存在する。

$S = [X \mapsto Q]T$ の場合

$S' = T$ とおけばよい。

$S = [Y \mapsto \mu Y.[X \mapsto Q]T']S_1$ で $S_1 \preceq [X \mapsto Q]T'$ の場合

帰納法の仮定を適用することで、二通りの場合が得られる。

- $S_1 \preceq Q$ 。束縛変数名に関する取り扱いから $Y \notin FV(Q)$ であるとわかっているので、$Y \notin FV(S_1)$ でなくてはならない。しかし、そうすると $S = [Y \mapsto \mu Y.[X \mapsto Q]T']S_1 = S_1$ であり $S \preceq Q$ となる。
- ある $S_2 \preceq T'$ が存在して $S_1 = [X \mapsto Q]S_2$ 。この場合 $S = [Y \mapsto \mu Y.[X \mapsto Q]T']S_1 = [Y \mapsto \mu Y.[X \mapsto Q]T'][X \mapsto Q]S_2 = [X \mapsto Q][Y \mapsto \mu Y.T']S_2$ である。$S' = [Y \mapsto \mu Y.T']S_2$ とおけば所望の結果が得られる。 □

証明の最後の部分は、μ 型の任意のトップダウン部分式を、ボトムアップ部分式の中から見つけられることを確立する。

命題 21.9.10. $S \sqsubseteq T$ ならば $S \preceq T$ □

[証明] $\mu TD \subseteq \mu BU$ であることを示したい。帰納法の原理により μBU が TD について閉じていることを示せればよい。これは $TD(\mu BU) \subseteq \mu BU$ ということである。すなわち、$(A, B) \in TD(\mu BU)$ ならば $(A, B) \in \mu BU = BU(\mu BU)$ であることを示したい。後者は、μBU から (A, B) を生成しうる TD の任意の節に、μBU から (A, B) を生成する BU の節が対応していれば真となる。これは TD の最後の節以外は、BU の対応する節と全く同じなので、自明に真である。最後の節では $(A, B) = (S, \mu X.T) \in TD(\mu BU)$ かつ $(S, [X \mapsto \mu X.T]T) \in \mu BU$（これは $S \preceq [X \mapsto \mu X.T]T$ と同値）である。補題 21.9.9 により $S \preceq \mu X.T$ もしくは、$(S', T) \in \mu BU$ なる S' が存在して $S = [X \mapsto \mu X.T]S'$ である。前者の場合は $(S, \mu X.T) \in \mu BU$ であり、これは求めていたものである。後者の場合は BU の最後の節により $(S, \mu X.T) \in BU(\mu BU) = \mu BU$ が含意される。 □

この節で証明してきた事実を組み合わせることで、最終的な結果が得られる。

命題 21.9.11. 任意の μ 型 S および T について、集合 $reachable_{S_m}(S, T)$ は有限である。 □

[証明] S と T に対して、Td をこれら二つのすべてのトップダウン部分式の集合とし、Bu をこれら二つのすべてのボトムアップ部分式の集合とする。命題 21.9.5 により、$reachable_{S_m}(S, T) \subseteq Td \times Td$ である。また、命題 21.9.10 により、$Td \times Td \subseteq Bu \times Bu$ である。補題 21.9.8 により、後者の集合は有限である。したがって、$reachable_{S_m}(S, T)$ は有限である。 □

21.10 余談：指数オーダーアルゴリズム

21.9 節冒頭で示したアルゴリズム *subtype*（図 21-4）は、新たな仮定の集合を返す代わりにブール値を返すようにすることで、もう少しだけ単純化できる（図 21-5 参照）。結果として得られる手続き $subtype^{ac}$ は、Amadio と Cardelli の部分型付けの検査アルゴリズム [18] に対応する。これは *subtype* が計算するのと同じ関係を計算するが、→ と × の場合の再帰呼び出しを通じて部分型関係に属する二つ組を記憶しないため、遥かに効率が悪い。この一見無害な変更により、このアルゴリ

$$
\begin{array}{lll}
\mathit{subtype}^{ac}(A,\mathsf{S},\mathsf{T}) & = & \text{if } (\mathsf{S},\mathsf{T}) \in A \text{ then } \mathit{true} \\
& & \text{else let } A_0 = A \cup (\mathsf{S},\mathsf{T}) \text{ in} \\
& & \quad \text{if } \mathsf{T} = \mathsf{Top} \text{ then } \mathit{true} \\
& & \quad \text{else if } \mathsf{S} = \mathsf{S}_1 \times \mathsf{S}_2 \text{ and } \mathsf{T} = \mathsf{T}_1 \times \mathsf{T}_2 \text{ then} \\
& & \qquad \mathit{subtype}^{ac}(A_0,\mathsf{S}_1,\mathsf{T}_1) \text{ and} \\
& & \qquad \mathit{subtype}^{ac}(A_0,\mathsf{S}_2,\mathsf{T}_2) \\
& & \quad \text{else if } \mathsf{S} = \mathsf{S}_1 \to \mathsf{S}_2 \text{ and } \mathsf{T} = \mathsf{T}_1 \to \mathsf{T}_2 \text{ then} \\
& & \qquad \mathit{subtype}^{ac}(A_0,\mathsf{T}_1,\mathsf{S}_1) \text{ and} \\
& & \qquad \mathit{subtype}^{ac}(A_0,\mathsf{S}_2,\mathsf{T}_2) \\
& & \quad \text{else if } \mathsf{S} = \mu\mathsf{X}.\mathsf{S}_1 \text{ then} \\
& & \qquad \mathit{subtype}^{ac}(A_0, [\mathsf{X} \mapsto \mu\mathsf{X}.\mathsf{S}_1]\mathsf{S}_1, \mathsf{T}) \\
& & \quad \text{else if } \mathsf{T} = \mu\mathsf{X}.\mathsf{T}_1 \text{ then} \\
& & \qquad \mathit{subtype}^{ac}(A_0, \mathsf{S}, [\mathsf{X} \mapsto \mu\mathsf{X}.\mathsf{T}_1]\mathsf{T}_1) \\
& & \quad \text{else } \mathit{false}
\end{array}
$$

▲ 図 21-5.　Amadio と Cardelli の部分型付けアルゴリズム

ズムが実行する再帰呼び出しの回数が爆発してしまう。*subtype* が行う再帰呼び出しの回数は、二つの引数の型の部分式の総数の二乗に比例する（このことは補題 21.9.8 と命題 21.9.11 の証明を精査することでわかる）のに対して、$\mathit{subtype}^{ac}$ の場合には指数オーダーになる。

以下の例で $\mathit{subtype}^{ac}$ の指数的な振る舞いがはっきりとわかる。型の族 S_n および T_n を以下のように帰納的に定義する。

$$
\begin{array}{lllllll}
\mathsf{S}_0 & = & \mu\mathsf{X}.\mathsf{Top}\times\mathsf{X} & \quad & \mathsf{S}_{n+1} & = & \mu\mathsf{X}.\mathsf{X}\to\mathsf{S}_n \\
\mathsf{T}_0 & = & \mu\mathsf{X}.\mathsf{Top}\times(\mathsf{Top}\times\mathsf{X}) & \quad & \mathsf{T}_{n+1} & = & \mu\mathsf{X}.\mathsf{X}\to\mathsf{T}_n
\end{array}
$$

S_n と T_n はそれぞれ S_{n-1} と T_{n-1} を一度ずつ含み、これらの（略記の展開後の）サイズは n に対して線形となる。しかし、以下の再帰呼び出しの列からわかるように、$\mathsf{S}_n <: \mathsf{T}_n$ の検査は指数オーダーの導出を生成する。

$$
\begin{array}{ll}
& \mathit{subtype}^{ac}(\varnothing,\mathsf{S}_n,\mathsf{T}_n) \\
= & \mathit{subtype}^{ac}(\mathsf{A}_1,\mathsf{S}_n\to\mathsf{S}_{n-1},\mathsf{T}_n) \\
= & \mathit{subtype}^{ac}(\mathsf{A}_2,\mathsf{S}_n\to\mathsf{S}_{n-1},\mathsf{T}_n\to\mathsf{T}_{n-1}) \\
= & \mathit{subtype}^{ac}(\mathsf{A}_3,\mathsf{T}_n,\mathsf{S}_n) \text{ and } \underline{\mathit{subtype}^{ac}(\mathsf{A}_3,\mathsf{S}_{n-1},\mathsf{T}_{n-1})} \\
= & \mathit{subtype}^{ac}(\mathsf{A}_4,\mathsf{T}_n\to\mathsf{T}_{n-1},\mathsf{S}_n) \text{ and } \ldots \\
= & \mathit{subtype}^{ac}(\mathsf{A}_5,\mathsf{T}_n\to\mathsf{T}_{n-1},\mathsf{S}_n\to\mathsf{S}_{n-1}) \text{ and } \ldots \\
= & \mathit{subtype}^{ac}(\mathsf{A}_6,\mathsf{S}_n,\mathsf{T}_n) \text{ and } \underline{\mathit{subtype}^{ac}(\mathsf{A}_6,\mathsf{T}_{n-1},\mathsf{S}_{n-1})} \text{ and } \ldots \\
= & \text{以下同様}
\end{array}
$$

ただし

$$
\begin{array}{lll}
\mathsf{A}_1 & = & \{(\mathsf{S}_n,\mathsf{T}_n)\} \\
\mathsf{A}_2 & = & \mathsf{A}_1 \cup \{(\mathsf{S}_n\to\mathsf{S}_{n-1},\mathsf{T}_n)\} \\
\mathsf{A}_3 & = & \mathsf{A}_2 \cup \{(\mathsf{S}_n\to\mathsf{S}_{n-1},\mathsf{T}_n\to\mathsf{T}_{n-1})\} \\
\mathsf{A}_4 & = & \mathsf{A}_3 \cup \{(\mathsf{T}_n,\mathsf{S}_n)\} \\
\mathsf{A}_5 & = & \mathsf{A}_4 \cup \{(\mathsf{T}_n\to\mathsf{T}_{n-1},\mathsf{S}_n)\} \\
\mathsf{A}_6 & = & \mathsf{A}_5 \cup \{(\mathsf{T}_n\to\mathsf{T}_{n-1},\mathsf{S}_n\to\mathsf{S}_{n-1})\}
\end{array}
$$

最初の呼び出し $subtype^{ac}(\emptyset, \mathsf{S}_n, \mathsf{T}_n)$ が、S_{n-1} と T_{n-1} を含む同じ形の、下線を引いた二つの再帰呼び出しを引き起こすことに注意されたい。これらは今度は S_{n-2} と T_{n-2} を含む二つの再帰呼び出しをそれぞれ引き起こし、以下同様である。したがって、再帰呼び出しの総数は 2^n に比例する。

21.11 同型再帰型の部分型付け

20.2 節で、再帰型を**同型再帰**で扱うことがあることを述べた。同型再帰では、再帰型の畳み込みと展開が項構築子 `fold` と `unfold` で明示的に証拠付けられている。そのような言語では、型中での μ 型構築子の位置によって、その型に属する項をどう使えるかが影響を受けるという意味で、μ 型構築子が「固い」。

同型再帰型を用いている言語に部分型付けを追加する際には、μ 型構築子の固さが部分型関係にも影響する。この節のほとんどでの「展開しきった極限で部分型を扱う」という直観的扱いの代わりに、再帰型にかかわる部分型関係を直接定義しなくてはならない。

同型再帰型部分型付けの最もよくある定義は **Amber 規則**というもので、これは Cardelli の Amber 言語 [74] で広まったためにこう呼ばれる。

$$\frac{\Sigma, \mathsf{X} <: \mathsf{Y} \vdash \mathsf{S} <: \mathsf{T}}{\Sigma \vdash \mu\mathsf{X}.\mathsf{S} <: \mu\mathsf{Y}.\mathsf{T}} \qquad \text{(S-AMBER)}$$

この規則は直観的には次のように解釈できる。「仮定（assumption）の集合 Σ の下で $\mu\mathsf{X}.\mathsf{S}$ が $\mu\mathsf{Y}.\mathsf{T}$ であることを示すには、追加の仮定 $\mathsf{X} <: \mathsf{Y}$ のもとで $\mathsf{S} <: \mathsf{T}$ を示せば十分である。」[†8] ここでの Σ は単に再帰変数の二つ組の集合であり、既に考慮してきている再帰型の二つ組を記録したものである。これらの仮定はもう一つの部分型付け規則

$$\frac{(\mathsf{X} <: \mathsf{Y}) \in \Sigma}{\Sigma \vdash \mathsf{X} <: \mathsf{Y}} \qquad \text{(S-ASSUMPTION)}$$

によって用いられる。この規則を用いることで、既に $\mathsf{X} <: \mathsf{Y}$ を仮定しているなら、それを結論することができる。

実質的には、これら二つの規則を第 16 章の通常の部分型付けアルゴリズムに追加（し他の規則を Σ を前提から結論に受け渡すように拡張）することで、図 21-5 の $subtype^{ac}$ アルゴリズムにやや似た振る舞いをするアルゴリズムが得られる。ここで Σ は $subtype^{ac}$ アルゴリズムの A の役割を果たす。違いは、(1) このアルゴリズムは <: の**両辺**に同時に再帰型が現れないと「展開」しない、(2) このアルゴリズムは再帰型をその本体へと置換しない（単に変数のままにしておく）という二点である。それによりアルゴリズムの停止性を確認しやすくなっている。

（第 19 章の Featherweight Java といった）**名前的**型システムにおける部分型付け規則は、Amber 規則と密接な関係がある。

演習 21.11.1. [推奨, ★★] 再帰型 S と T で、同値再帰的な型付けでは $\mathsf{S} <: \mathsf{T}$ だが、Amber 規則ではそうではないものを見つけよ。 □

†8 この規則は、両辺に束縛構文があるほとんどの規則（例えば図 26-1 の S-ALL）と異なり、規則の適用前に束縛変数 X と Y の名前を**相異なる**ように付け替える必要がある。

21.12 注記

この章は Gapeyev, Levin, and Pierce [168] のチュートリアル記事に基づいている。

余帰納法に関する背景は Barwise と Moss による "*Vicious Circles*" [42]、Gordon による余帰納法と関数的プログラミングのチュートリアル [181]、Milner と Tofte によるプログラミング言語意味論における余帰納法の解説記事 [313] にある。単調関数と不動点に関する基本的な情報については、Aczel [13] と Davey and Priestley [131] を参照せよ。

計算機科学における余帰納的な証明手法の使用は、1970 年代から始まる。例えば、並行性に関する Milner [308] と Park [351] の業績や、Arbib と Manes によるオートマトン理論における双対性の圏論的な議論 [21] を見よ。しかし、双対として「余」の付いた形の帰納法は、相当古くから数学者たちにとっては馴染み深いものであり、例えば普遍代数や圏論といった領域では明示的に展開されていた。非整礎集合に関する Aczel の名著 [14] は、歴史に関する簡潔なサーベイを含んでいる。

Amadio and Cardelli [18] は再帰型の部分型付けの最初のアルゴリズムを与えた。彼らの論文は次の三つの関係を定義している。無限木型の間の包含関係、μ 型の間の部分型関係を検査するアルゴリズム（によって定義される関係)、宣言的な推論規則の集合の最小不動点として定義した μ 型間の基準となる部分型関係である。これらはすべて等価であることが証明され、部分同値関係に基づいたモデルの構築につながっている。余帰納法は使わず、無限木について推論するためには、代わりに無限の木に対する有限の近似という概念を導入している。この概念は多くの証明で重要な役割を果たす。

Brandt and Henglein [56] は、Amadio と Cardelli の体系の根底にある余帰納的な本質を明らかにし、Amadio と Cardelli の部分型関係に対して健全かつ完全な新たな部分型関係を帰納的に公理化した。その公理化で Arrow/Fix と呼ばれる規則は、体系の余帰納性を具体化している。この論文は、余帰納法によって自然に定義される関係に対して帰納的な公理化を導出するための一般的な手法を述べており、また部分型付けアルゴリズムの停止性の詳細な証明も示している。この章の 21.9 節はその証明を綿密に追っている。Brandt と Henglein は彼らのアルゴリズムの計算量が $O(n^2)$ であることを証明した。

Kozen, Palsberg, and Schwartzbach [266] は、正則な再帰型はラベル付きの状態を持つオートマトンに対応するという観察に基づき、洗練された二乗オーダーの部分型付けアルゴリズムを得た。彼らの定義した二つのオートマトンの積は、通常の語のオートマトンで、元の二つのオートマトンが部分型関係にないとき、かつそのときに限り、語を受理する。そうすることで、受理言語が空であるかどうかについての線形時間での検査を用いて部分型付け問題が解ける。この事実と、直積の構成の二乗オーダーの計算量と、型からオートマトンへの線形時間の計算量から、全体が二乗オーダーの計算量であることがわかる。

Hosoya, Vouillon, and Pierce [222] は関連するオートマトン理論的なアプローチを用い、XML 処理アプリケーションに特化した部分型付けアルゴリズムにおいて、木オートマトンと（合併型付きの）再帰型を関連付けた。

Jim and Palsberg [240] は部分型と再帰型を持つ言語の型**再構築**（第 22 章参照） に取り組んだ。この章で我々がしたように、彼らは無限木上の部分型関係の余帰納的な見方を採用し、与えられた型の二つ組から極小な模倣関係（すなわち我々の用語での整合的な集合）を構築する手続きとしての部分型付け検査アルゴリズムを動機付けている。彼らは型上の関係の整合性と $P1$ 閉包の概念を定義した。これらは我々の整合性と到達可能集合に対応する。

それについて十分に長く考えれば、それは自明だと思うだろう。 —*Saul Gorn*

第V部

多相性

第 22 章

型再構築

本章で扱う体系は、単純型付きラムダ計算（図 9-1）にブール値（図 8-1）、数（図 8-2）、無限個の基本型（図 11-1）を加えたものである。対応する OCaml 実装は `recon` と `fullrecon` である。

これまで見てきた計算体系の型検査アルゴリズムは、すべて明示的な型注釈に依存していた。具体的には、ラムダ抽象の引数の型を注釈してやる必要があった。本章では、より強力な**型再構築**アルゴリズムを説明する。このアルゴリズムを用いると、一部またはすべての型注釈が指定されていない項の**主要型**を求められる。ML や Haskell などの言語の中心的存在として、同種のアルゴリズムが用いられている。

型再構築と他の言語機能を組み合わせることは、いささかデリケートな問題となりやすい。特に、レコードと部分型は重大な課題をもたらす。話を簡単にするために、ここでは単純型だけの型再構築を考える。他の組み合わせについて詳しく知りたければ、22.8 節にある文献を辿っていくとよい。

22.1 型変数と型代入

これまでの章で紹介した計算体系の中には、型の集合が**非解釈**な無限個の基本型（11.1 節）を含むことを仮定しているものがあった。Bool や Nat のような解釈のある基本型とは異なり、非解釈な基本型は項の導入や除去の操作を持たない。つまり直観的には、何か特定の型のプレースホルダーにすぎない（その型の実体は何でもよい）。本章では、「項 t のプレースホルダー X を具体的な型 Bool で具体化すると、型付け可能な項が得られるかどうか」というような問いについて考える。つまり、他の型を**代入**（または**具体化**）できる**型変数**として非解釈な基本型を扱う。

本章での技術的説明のためには、型を型変数に代入する操作を二つに分けると都合がよい。型変数から型への写像 σ（**型代入**といわれる）を**定義**することと、この対応を具体的な型 T に**適用**して具体化 σT を得ることである。例えば σ = [X ↦ Bool] を定義して、それから σ を型 X→X に適用すると、σ(X→X) = Bool→Bool を得る。

定義 22.1.1. 形式的には、**型代入**（または、型について話していることが明らかな場合には単に**代入**とも）は、型変数から型への有限写像である。例えば、X を T に、Y を U に対応付ける代入のことを [X ↦ T, Y ↦ U] と書く。σ に含まれる組の左側に現れる型変数の集合を $dom(\sigma)$ と書き、右側に現れる型の集合を $range(\sigma)$ と書く。代入の定義域（*dom*）と値域（*range*）に同時に同じ変数が現れうることに注意されたい。そのような場合、項の代入と同じように、代入のすべての節が同時に適用されることを意図している。例えば、[X ↦ Bool, Y ↦ X→X] は X を Bool に対応付け、Y を X→X に対応付ける。Y を Bool→Bool に対応付けるのではないことに注意されたい。

型に対する代入の適用は、自明な方法で定義される。

$$
\begin{array}{lcl}
\sigma(\mathtt{X}) & = & \begin{cases} \mathtt{T} & (\mathtt{X} \mapsto \mathtt{T}) \in \sigma \text{ の場合} \\ \mathtt{X} & \mathtt{X} \text{ が } \sigma \text{ の定義域にない場合} \end{cases} \\
\sigma(\mathtt{Nat}) & = & \mathtt{Nat} \\
\sigma(\mathtt{Bool}) & = & \mathtt{Bool} \\
\sigma(\mathtt{T}_1 \to \mathtt{T}_2) & = & \sigma\mathtt{T}_1 \to \sigma\mathtt{T}_2
\end{array}
$$

型代入時の変数捕獲を避けるために特別に何かをする必要はないことに注意されたい。これは、型変数を**束縛**する構文が型式の言語に備わっていないためである（そのような構文は第 23 章で扱う）。

型代入は、文脈に対しても各要素への適用として拡張される。

$$\sigma(\mathsf{x}_1{:}\mathsf{T}_1,\ldots,\mathsf{x}_n{:}\mathsf{T}_n) = (\mathsf{x}_1{:}\sigma\mathsf{T}_1,\ldots,\mathsf{x}_n{:}\sigma\mathsf{T}_n)$$

同様に、代入は項 t に対しても適用される。t 中の型注釈に現れるすべての型に対して代入を適用すればよい。

σ と γ が代入であるとき、それらを以下のように合成して得られる代入を $\sigma \circ \gamma$ と書く。

$$\sigma \circ \gamma = \left[\begin{array}{ll} \mathsf{X} \mapsto \sigma(\mathsf{T}) & (\mathsf{X} \mapsto \mathsf{T}) \in \gamma \text{ である場合} \\ \mathsf{X} \mapsto \mathsf{T} & (\mathsf{X} \mapsto \mathsf{T}) \in \sigma \text{ であり、} \mathsf{X} \notin \mathit{dom}(\gamma) \text{ である場合} \end{array}\right]$$

$(\sigma \circ \gamma)\mathsf{S} = \sigma(\gamma\mathsf{S})$ であることに注意されたい。 □

型代入の重要な性質として、型付け判断式の妥当性を保存するということがある。つまり、型変数を含む項が正しく型付けされているならば、その項の代入による具体化もすべて正しく型付けされたものになる。

定理 22.1.2.［型代入の下での型付けの保存］ σ が任意の型代入で、かつ $\Gamma \vdash \mathsf{t} : \mathsf{T}$ ならば、$\sigma\Gamma \vdash \sigma\mathsf{t} : \sigma\mathsf{T}$ である。 □

［証明］ 型付け導出に関する単純な帰納法。 □

22.2 型変数についての二つの視点

t を型変数を含む項とし、Γ を関連する文脈（これも型変数を含みうる）とする。t に関して、大きく異なる二つの問いがある。

(1) 「t の**すべての**代入による具体化は正しく型付けされるだろうか」、つまり、すべての σ に対して、$\sigma\Gamma \vdash \sigma\mathsf{t} : \mathsf{T}$ なる T が存在するだろうか。
(2) 「t の**ある**代入による具体化は正しく型付けされるだろうか」、つまり、$\sigma\Gamma \vdash \sigma\mathsf{t} : \mathsf{T}$ なる T が存在するような σ を見つけられるだろうか。

一つ目の視点によると、型変数は型検査の間は**抽象的なままである**べきである。これにより、正しく型付けされた項は、後で型変数にいかなる具体的な型を代入したとしても適切に振る舞うことが保証される。例えば、項

```
λf:X→X. λa:X. f (f a);
```

は型 (X→X)→X→X を持つ。X を具体的な型 T に置き換えたときはいつでも、その具体化

```
λf:T→T. λa:T. f (f a);
```

は正しく型付けされる。このように型変数を抽象的なままにすることは、**パラメータ多相**につながる。パラメータ多相では、一つの項が複数の具体的な文脈で、さまざまな具体型を持つものとして使用できるということを表現するために型変数が使われる。パラメータ多相については本章でのちに触れる（22.7 節）。より詳しくは第 23 章で説明する。

二つ目の視点では、元々の項 t はそもそも正しく型付けされていないかもしれない。知りたいのは、幾つかの型変数に適切な値を選ぶことで、その項を正しく型付けされた項に**具体化**できるかど

うかである。例えば、項

λf:Y. λa:X. f (f a);

はこのままでは型付け可能でない。しかし Y を Nat→Nat で置き換え、X を Nat で置き換えたら、項

λf:Nat→Nat. λa:Nat. f (f a);

となり、これは型 (Nat→Nat)→Nat→Nat を持つ。または、単に Y を X→X に置き換えるだけでも、項

λf:X→X. λa:X. f (f a);

が得られる。これは型変数を残しているが、正しく型付けされる。実際、この項は λf:Y. λa:X. f (f a) の**最も一般的な**具体化である。つまり、正しく型付けされた項を得るために、型変数の値について最小限の約束しかしないという意味である。

型変数の妥当な具体化を探すことは、**型再構築**（しばしば**型推論**とも）のアイディアにつながる。このアイディアの下では、プログラマが省略した型情報の一部をコンパイラが埋めていく。極端な話、ML のようにプログラマが型注釈を**すべて**省略し、素の型無しラムダ計算の文法で書くことも可能である。構文解析中に、素のラムダ抽象 λx.t それぞれに型変数の注釈を付けて λx:X.t とする。このとき X はプログラム中の他のどの抽象とも異なる型変数から選ぶ。それから型再構築を行い、すべての変数に対して、項の型検査が成功するような最も一般的な値を見つける。（この話は、ML の let 多相が存在すると少し複雑になる。この話の続きは 22.6 節と 22.7 節で行う。）

型再構築を形式化するためには、項や関連する文脈の中の型変数への代入のうち、妥当な型付け判断式が得られるようなものについて、簡潔に述べる手段が必要である[†1]。

定義 22.2.1. Γ を文脈とし、t を項とする。$(\Gamma, \mathtt{t})$ の**解**とは、$\sigma\Gamma \vdash \sigma\mathtt{t} : \mathtt{T}$ なる組 $(\sigma, \mathtt{T})$ のことである。 □

例 22.2.2. $\Gamma = \mathtt{f{:}X, a{:}Y}$ かつ $\mathtt{t} = \mathtt{f\ a}$ とする。このとき、

([X ↦ Y→Nat], Nat)　　([X ↦ Y→Z], Z)
([X ↦ Y→Z, Z ↦ Nat], Z)　　([X ↦ Y→Nat→Nat], Nat→Nat)
([X ↦ Nat→Nat, Y ↦ Nat], Nat)

はすべて $(\Gamma, \mathtt{t})$ の解である。 □

演習 22.2.3. [★ ↛]　空の文脈における以下の項の異なる解を三つ求めよ。

λx:X. λy:Y. λz:Z. (x z) (y z)

□

22.3 制約に基づいた型付け

では、項 t と文脈 Γ が与えられたとき、$(\Gamma, \mathtt{t})$ の任意の解によって充足されなければならない制約の集合を求めるアルゴリズムを示す。この制約は、（型変数を含みうる）型式の間の等式である。このアルゴリズムの裏にある直観は、本質的には通常の型検査アルゴリズムと同じである。唯一異

[†1] これらの基本的な定義を与える方法は他にもある。一つは、Kirchner and Jouannaud [263] に従い、図 22-1 の制約生成規則にあるすべてのフレッシュさの条件の代わりに、**存在型被単一化子**と呼ばれる一般的な仕組みを用いる方法である。他の可能な改良には、Rémy [401; 402]（長い版は [405] の Chapter 5）が用いた、型付け判断式自身を被単一化子として扱う方法がある。三つ組 $(\Gamma, \mathtt{t}, \mathtt{T})$ から始め（三つの要素はどれも型変数を含みうる）、$\sigma\Gamma \vdash \sigma(\mathtt{t}) : \sigma(\mathtt{T})$ となるような代入 σ、つまりスキーマ的な型付け判断式 $\Gamma \vdash \mathtt{t} : \mathtt{T}$ を**単一化**する代入を探す。

$$\frac{\mathtt{x}{:}\mathsf{T} \in \Gamma}{\Gamma \vdash \mathtt{x} : \mathsf{T} \;|_{\varnothing}\; \{\}} \quad \text{(CT-Var)}$$

$$\frac{\Gamma, \mathtt{x}{:}\mathsf{T}_1 \vdash \mathtt{t}_2 : \mathsf{T}_2 \;|_{\mathcal{X}}\; C}{\Gamma \vdash \lambda \mathtt{x}{:}\mathsf{T}_1 . \mathtt{t}_2 : \mathsf{T}_1 \to \mathsf{T}_2 \;|_{\mathcal{X}}\; C} \quad \text{(CT-Abs)}$$

$$\frac{\begin{array}{c}\Gamma \vdash \mathtt{t}_1 : \mathsf{T}_1 \;|_{\mathcal{X}_1}\; C_1 \qquad \Gamma \vdash \mathtt{t}_2 : \mathsf{T}_2 \;|_{\mathcal{X}_2}\; C_2 \\ \mathcal{X}_1 \cap \mathcal{X}_2 = \mathcal{X}_1 \cap FV(\mathsf{T}_2) = \mathcal{X}_2 \cap FV(\mathsf{T}_1) = \varnothing \\ \mathsf{X} \text{ は } \mathcal{X}_1, \mathcal{X}_2, \mathsf{T}_1, \mathsf{T}_2, C_1, C_2, \Gamma, \mathtt{t}_1, \mathtt{t}_2 \text{ のいずれにも現れない} \\ C' = C_1 \cup C_2 \cup \{\mathsf{T}_1 = \mathsf{T}_2 \to \mathsf{X}\}\end{array}}{\Gamma \vdash \mathtt{t}_1\ \mathtt{t}_2 : \mathsf{X} \;|_{\mathcal{X}_1 \cup \mathcal{X}_2 \cup \{\mathsf{X}\}}\; C'} \quad \text{(CT-App)}$$

$$\Gamma \vdash \mathtt{0} : \mathsf{Nat} \;|_{\varnothing}\; \{\} \quad \text{(CT-Zero)}$$

$$\frac{\begin{array}{c}\Gamma \vdash \mathtt{t}_1 : \mathsf{T} \;|_{\mathcal{X}}\; C \\ C' = C \cup \{\mathsf{T} = \mathsf{Nat}\}\end{array}}{\Gamma \vdash \mathtt{succ}\ \mathtt{t}_1 : \mathsf{Nat} \;|_{\mathcal{X}}\; C'} \quad \text{(CT-Succ)}$$

$$\frac{\begin{array}{c}\Gamma \vdash \mathtt{t}_1 : \mathsf{T} \;|_{\mathcal{X}}\; C \\ C' = C \cup \{\mathsf{T} = \mathsf{Nat}\}\end{array}}{\Gamma \vdash \mathtt{pred}\ \mathtt{t}_1 : \mathsf{Nat} \;|_{\mathcal{X}}\; C'} \quad \text{(CT-Pred)}$$

$$\frac{\begin{array}{c}\Gamma \vdash \mathtt{t}_1 : \mathsf{T} \;|_{\mathcal{X}}\; C \\ C' = C \cup \{\mathsf{T} = \mathsf{Nat}\}\end{array}}{\Gamma \vdash \mathtt{iszero}\ \mathtt{t}_1 : \mathsf{Bool} \;|_{\mathcal{X}}\; C'} \quad \text{(CT-IsZero)}$$

$$\Gamma \vdash \mathtt{true} : \mathsf{Bool} \;|_{\varnothing}\; \{\} \quad \text{(CT-True)}$$

$$\Gamma \vdash \mathtt{false} : \mathsf{Bool} \;|_{\varnothing}\; \{\} \quad \text{(CT-False)}$$

$$\frac{\begin{array}{c}\Gamma \vdash \mathtt{t}_1 : \mathsf{T}_1 \;|_{\mathcal{X}_1}\; C_1 \\ \Gamma \vdash \mathtt{t}_2 : \mathsf{T}_2 \;|_{\mathcal{X}_2}\; C_2 \qquad \Gamma \vdash \mathtt{t}_3 : \mathsf{T}_3 \;|_{\mathcal{X}_3}\; C_3 \\ \mathcal{X}_1, \mathcal{X}_2, \mathcal{X}_3 \text{ はいずれも重複しない} \\ C' = C_1 \cup C_2 \cup C_3 \cup \{\mathsf{T}_1 = \mathsf{Bool}, \mathsf{T}_2 = \mathsf{T}_3\}\end{array}}{\Gamma \vdash \mathtt{if}\ \mathtt{t}_1\ \mathtt{then}\ \mathtt{t}_2\ \mathtt{else}\ \mathtt{t}_3 : \mathsf{T}_2 \;|_{\mathcal{X}_1 \cup \mathcal{X}_2 \cup \mathcal{X}_3}\; C'} \quad \text{(CT-If)}$$

▲ 図 22-1.　制約型付け規則

なるのは、制約を**検査**するのではなく、後の検討のために**記録**するだけということである。例えば、$\Gamma \vdash \mathtt{t}_1 : \mathsf{T}_1$ かつ $\Gamma \vdash \mathtt{t}_2 : \mathsf{T}_2$ を満たす適用 $\mathtt{t}_1\ \mathtt{t}_2$ があるとき、T_1 が $\mathsf{T}_2 \to \mathsf{T}_{12}$ の形であることを検査し T_{12} を適用の型として返すのではなく、フレッシュな型変数 X を選び、制約 $\mathsf{T}_1 = \mathsf{T}_2 \to \mathsf{X}$ を記録し、X を適用の型として返すのである。

定義 22.3.1.　制約集合 C は等式の集合 $\{\mathsf{S}_i = \mathsf{T}_i\ ^{i \in 1..n}\}$ である。S と T の代入による具体化 $\sigma\mathsf{S}$ と $\sigma\mathsf{T}$ が同一になるとき、代入 σ は等式 S = T を**単一化**するという。σ が C のすべての等式を単一化するとき、σ は C を**単一化**する（または**充足**する）という。 □

定義 22.3.2.　制約型付け関係 $\Gamma \vdash \mathtt{t} : \mathsf{T} \;|_{\mathcal{X}}\; C$ は図 22-1 の規則で定義される。非形式的には、$\Gamma \vdash \mathtt{t} : \mathsf{T} \;|_{\mathcal{X}}\; C$ は「制約 C が充足されるならば、項 t は Γ の前提の下で型 T を持つ」と読める。規則 CT-App において、$FV(\mathsf{T})$ は T に現れるすべての型変数の集合を表す。

下付き文字の $\mathcal{X}$ を用いて、各部分導出において導入される型変数を追えるようにし、異なる部分導出で作られるフレッシュな変数が実際に異なるものであることを保証している。この規則を最初に読むときは、この下付き文字と、それを含む前提をすべて無視するとわかりやすいかもしれない。次に読むときは、これらの注釈と前提が二つのことを保証していることを観察せよ。一つは、ある導出の最後の規則で型変数が選ばれたときは必ず、その型変数は部分導出の中で選ばれたどの変数とも異なるものになるということである。もう一つは、規則が二つ以上の部分導出を含むときは必ず、各部分導出で選ばれた変数の集合がいずれも重複しないということである。また、これらの条件が、与えられた項に対する導出の構築**そのもの**を妨げはしないことに注意されたい。これらの条件は単に、異なる箇所で同じ変数を「フレッシュ」なものとして用いた導出を構築してしまわないようにしているだけである。型変数名は無限にあるので、このフレッシュさの要求を充足する方法は必ず見つかる。

制約型付け規則を下から上に読むと、Γ と t が与えられたとき $\Gamma \vdash \mathtt{t} : \mathtt{T} \mid_{\mathcal{X}} C$ なる T と C（そして $\mathcal{X}$）を計算する単純な手続きとなる。しかし単純型付きラムダ計算の通常の型付けアルゴリズムと違って、この手続きは決して失敗しない。すべての Γ と t に対して $\Gamma \vdash \mathtt{t} : \mathtt{T} \mid_{\mathcal{X}} C$ なる T と C が必ず存在し、さらに T と C は Γ と t から一意に決まるためである。（厳密に言うと、このアルゴリズムは「フレッシュな名前の選択の違いを無視」する場合に限り、決定的といえる。このことは演習 22.3.9 で振り返る。）

以下の議論では、記法を簡単にするため、$\mathcal{X}$ を省略して単に $\Gamma \vdash \mathtt{t} : \mathtt{T} \mid C$ と書くことがある。

□

演習 22.3.3. [★ ↛] ある S、$\mathcal{X}$、C に対して、結論が以下になるような制約型付け導出を構築せよ。

$$\vdash \lambda\mathtt{x}{:}\mathtt{X}.\ \lambda\mathtt{y}{:}\mathtt{Y}.\ \lambda\mathtt{z}{:}\mathtt{Z}.\ (\mathtt{x}\ \mathtt{z})\ (\mathtt{y}\ \mathtt{z}) : \mathtt{S} \mid_{\mathcal{X}} C$$

□

制約型付け関係のアイディアは、項 t と文脈 Γ が与えられたとき、t が Γ の下で型付け可能かどうかを、以下の要領で検査できるというものである。まず、t が型を持つために充足されるべき制約 C を集め、同時に結果の型 S を求める。ここで S は、C と型変数を共有し、その型変数の観点から t が持ちうる型を特徴付ける型である。その後は単に、t の解を見つけるために、C を充足する（すなわち、C 内のすべての等式の両辺を同一にする）代入 σ を探す。そのような任意の σ に対して、型 σS は t が持ちうる型になる。もし C を充足する代入が**存在しない**と判明した場合、型付け可能にするように t を具体化することはできないとわかる。

例えば、項 $\mathtt{t} = \lambda\mathtt{x}{:}\mathtt{X}{\to}\mathtt{Y}.\ \mathtt{x}\ 0$ に対してアルゴリズムが生成する制約集合は $\{\mathtt{X}{\to}\mathtt{Y} = \mathtt{Nat}{\to}\mathtt{Z}\}$ であり、これに対応する結果の型は $(\mathtt{X}{\to}\mathtt{Y}){\to}\mathtt{Z}$ である。代入 $\sigma = [\mathtt{X} \mapsto \mathtt{Nat}, \mathtt{Z} \mapsto \mathtt{Bool}, \mathtt{Y} \mapsto \mathtt{Bool}]$ は式 $\mathtt{X}{\to}\mathtt{Y} = \mathtt{Nat}{\to}\mathtt{Z}$ の両辺を同一にするので、$\sigma((\mathtt{X}{\to}\mathtt{Y}){\to}\mathtt{Z})$、つまり $(\mathtt{Nat}{\to}\mathtt{Bool}){\to}\mathtt{Bool}$ は t が持ちうる型であるとわかる。

この考え方は以下の定義で形式化できる。

定義 22.3.4. $\Gamma \vdash \mathtt{t} : \mathtt{S} \mid C$ とする。$(\Gamma, \mathtt{t}, \mathtt{S}, C)$ の**解**とは、σ が C を充足し、かつ $\sigma\mathtt{S} = \mathtt{T}$ となるような組 $(\sigma, \mathtt{T})$ のことである。 □

与えられた制約集合 C を単一化する代入を見つけるアルゴリズム的な問題は、次節で取り上げる。しかしまずは、制約型付けアルゴリズムが元々の宣言的な型付け関係と適切な意味で対応することを確認すべきである。

文脈 Γ と項 t が与えられたとき、Γ と t の型変数を具体化して妥当な型付けを生成する方法には、二種類の表現がある。

(1) ［宣言的な方法］定義 22.2.1 の意味で、$(\Gamma, \mathtt{t})$ のすべての解の集合として表現する。
(2) ［アルゴリズム的な方法］制約型付け関係を通して、$\Gamma \vdash \mathtt{t} : \mathtt{S} \mid C$ を満たす S と C を見つけ、それから $(\Gamma, \mathtt{t}, \mathtt{S}, C)$ の解の集合を得る。

この二つの表現方法が等価であることを次の 2 段階で示す。まず、$(\Gamma, \mathtt{t}, \mathtt{S}, C)$ の解はすべて $(\Gamma, \mathtt{t})$ の解でもあることを示す（定理 22.3.5）。そして $(\Gamma, \mathtt{t})$ の解はすべて、制約生成で導入された型変数に値を与えることで $(\Gamma, \mathtt{t}, \mathtt{S}, C)$ の解に**拡張**できることを示す（定理 22.3.7）。

定理 22.3.5.［制約型付けの健全性］ $\Gamma \vdash \mathtt{t} : \mathtt{S} \mid C$ とする。もし $(\sigma, \mathtt{T})$ が $(\Gamma, \mathtt{t}, \mathtt{S}, C)$ の解であれば、それは $(\Gamma, \mathtt{t})$ の解でもある。 □

この議論をする上では、フレッシュな変数集合 $\mathcal{X}$ は重要ではないので省略できる。

［証明］ $\Gamma \vdash \mathtt{t} : \mathsf{S} \mid C$ の制約型付け導出に関する帰納法による。最後に使われた規則に基づく場合分けを行う。

CT-VAR の場合　$\mathtt{t} = \mathtt{x}$　　$\mathtt{x}{:}\mathsf{S} \in \Gamma$　　$C = \{\}$

仮定より、(σ, T) は $(\Gamma, \mathtt{t}, \mathsf{S}, C)$ の解である。C は空であるから、これは単に $\sigma\mathsf{S} = \mathsf{T}$ を意味する。しかしそれならば、T-VAR により、直ちに $\sigma\Gamma \vdash \mathtt{x} : \mathsf{T}$ を得る。これは所要の通りである。

CT-ABS の場合　$\mathtt{t} = \lambda\mathtt{x}{:}\mathsf{T}_1.\mathtt{t}_2$　　$\mathsf{S} = \mathsf{T}_1 \to \mathsf{S}_2$　　$\Gamma, \mathtt{x}{:}\mathsf{T}_1 \vdash \mathtt{t}_2 : \mathsf{S}_2 \mid C$

仮定より、(σ, T) は $(\Gamma, \mathtt{t}, \mathsf{S}, C)$ の解である。つまり、σ は C を単一化し、$\mathsf{T} = \sigma\mathsf{S} = \sigma\mathsf{T}_1 \to \sigma\mathsf{S}_2$ である。よって、$(\sigma, \sigma\mathsf{S}_2)$ は $((\Gamma, \mathtt{x}{:}\mathsf{T}_1), \mathtt{t}_2, \mathsf{S}_2, C)$ の解である。帰納法の仮定により、$(\sigma, \sigma\mathsf{S}_2)$ は $((\Gamma,\ \mathtt{x}{:}\mathsf{T}_1),\ \mathtt{t}_2)$ の解である。つまり $\sigma\Gamma,\ \mathtt{x}{:}\sigma\mathsf{T}_1 \vdash \sigma\mathtt{t}_2 : \sigma\mathsf{S}_2$ である。T-ABS により、$\sigma\Gamma \vdash \lambda\mathtt{x}{:}\sigma\mathsf{T}_1.\sigma\mathtt{t}_2 : \sigma\mathsf{T}_1 \to \sigma\mathsf{S}_2 = \sigma(\mathsf{T}_1 \to \mathsf{S}_2) = \mathsf{T}$ となる。これは所要の通りである。

CT-APP の場合　$\mathtt{t} = \mathtt{t}_1\ \mathtt{t}_2$　　$\mathsf{S} = \mathsf{X}$

$\Gamma \vdash \mathtt{t}_1 : \mathsf{S}_1 \mid C_1$　　$\Gamma \vdash \mathtt{t}_2 : \mathsf{S}_2 \mid C_2$

$C = C_1 \cup C_2 \cup \{\mathsf{S}_1 = \mathsf{S}_2 \to \mathsf{X}\}$

定義より、σ は C_1 と C_2 を単一化し、かつ $\sigma\mathsf{S}_1 = \sigma(\mathsf{S}_2 \to \mathsf{X})$ である。よって、$(\sigma,\ \sigma\mathsf{S}_1)$ と $(\sigma,\ \sigma\mathsf{S}_2)$ は $(\Gamma, \mathtt{t}_1, \mathsf{S}_1, C_1)$ と $(\Gamma, \mathtt{t}_2, \mathsf{S}_2, C_2)$ の解である。このことから、帰納法の仮定により $\sigma\Gamma \vdash \sigma\mathtt{t}_1 : \sigma\mathsf{S}_1$ かつ $\sigma\Gamma \vdash \sigma\mathtt{t}_2 : \sigma\mathsf{S}_2$ が得られる。しかし $\sigma\mathsf{S}_1 = \sigma\mathsf{S}_2 \to \sigma\mathsf{X}$ であるから、$\sigma\Gamma \vdash \sigma\mathtt{t}_1 : \sigma\mathsf{S}_2 \to \sigma\mathsf{X}$ を得る。T-APP により、$\sigma\Gamma \vdash \sigma(\mathtt{t}_1\ \mathtt{t}_2) : \sigma\mathsf{X} = \mathsf{T}$ である。

その他の場合

同様に示せる。 □

通常の型付け関係に対する制約型付けの**完全性**の議論は、もう少し注意が必要である。フレッシュな名前を慎重に扱わなければならないためである。

定義 22.3.6. $\mathcal{X}$ に含まれる変数のすべてに対して未定義で、他の変数には σ と同じように振る舞う代入を、$\sigma \backslash \mathcal{X}$ と書く。 □

定理 22.3.7.［制約型付けの完全性］ $\Gamma \vdash \mathtt{t} : \mathsf{S}\ \ |_{\mathcal{X}}\ C$ とする。(σ, T) が $(\Gamma, \mathtt{t})$ の解であり、かつ $dom(\sigma) \cap \mathcal{X} = \emptyset$ ならば、$\sigma' \backslash \mathcal{X} = \sigma$ となるような $(\Gamma, \mathtt{t}, \mathsf{S}, C)$ の解 (σ', T) が存在する。 □

［証明］ 制約型付け導出に関する帰納法による。

CT-VAR の場合　$\mathtt{t} = \mathtt{x}$　　$\mathtt{x}{:}\mathsf{S} \in \Gamma$

(σ, T) が $(\Gamma, \mathtt{x})$ の解であるという仮定から、型付け関係の逆転補題（補題 9.3.1）より、$\mathsf{T} = \sigma\mathsf{S}$ とわかる。しかしそれならば、(σ, T) は $(\Gamma, \mathtt{x}, \mathsf{S}, \{\})$ の解でもある。

CT-ABS の場合　$\mathtt{t} = \lambda\mathtt{x}{:}\mathsf{T}_1.\mathtt{t}_2$　　$\Gamma, \mathtt{x}{:}\mathsf{T}_1 \vdash \mathtt{t}_2 : \mathsf{S}_2\ \ |_{\mathcal{X}}\ C$　　$\mathsf{S} = \mathsf{T}_1 \to \mathsf{S}_2$

(σ, T) は $(\Gamma,\ \lambda\mathtt{x}{:}\mathsf{T}_1.\mathtt{t}_2)$ の解であるという仮定から、型付け関係の逆転補題より、ある T_2 に対して $\sigma\Gamma, \mathtt{x}{:}\sigma\mathsf{T}_1 \vdash \sigma\mathtt{t}_2 : \mathsf{T}_2$ かつ $\mathsf{T} = \sigma\mathsf{T}_1 \to \mathsf{T}_2$ となる。帰納法の仮定により、$((\Gamma,\mathtt{x}{:}\mathsf{T}_1), \mathtt{t}_2, \mathsf{S}_2, C)$ に対して $\sigma' \backslash \mathcal{X}$ が σ と同じになるような解 (σ', T_2) が存在する。ここで、$\mathcal{X}$ は T_1 に含まれる型変数をいずれも含むことはできない。よって、$\sigma'\mathsf{T}_1 = \sigma\mathsf{T}_1$ かつ $\sigma'(\mathsf{S}) = \sigma'(\mathsf{T}_1 \to \mathsf{S}_2) = \sigma\mathsf{T}_1 \to \sigma'\mathsf{S}_2 = \sigma\mathsf{T}_1 \to \mathsf{T}_2 = \mathsf{T}$ である。したがって、(σ', T) が $(\Gamma, (\lambda\mathtt{x}{:}\mathsf{T}_1.\mathtt{t}_2), \mathsf{T}_1 \to \mathsf{S}_2, C)$ の解であるとわかる。

CT-App の場合 $\mathsf{t} = \mathsf{t}_1\ \mathsf{t}_2 \qquad \Gamma \vdash \mathsf{t}_1 : \mathsf{S}_1 \ |_{\mathcal{X}_1}\ C_1 \qquad \Gamma \vdash \mathsf{t}_2 : \mathsf{S}_2 \ |_{\mathcal{X}_2}\ C_2$

$\mathcal{X}_1 \cap \mathcal{X}_2 = \emptyset$

$\mathcal{X}_1 \cap FV(\mathsf{S}_2) = \emptyset$

$\mathcal{X}_2 \cap FV(\mathsf{S}_1) = \emptyset$

X が $\mathcal{X}_1, \mathcal{X}_2, \mathsf{S}_1, \mathsf{S}_2, C_1, C_2, \Gamma, \mathsf{t}_1, \mathsf{t}_2$ に現れない

$\mathsf{S} = \mathsf{X} \qquad \mathcal{X} = \mathcal{X}_1 \cup \mathcal{X}_2 \cup \{\mathsf{X}\} \qquad C = C_1 \cup C_2 \cup \{\mathsf{S}_1 = \mathsf{S}_2 \to \mathsf{X}\}$

(σ, T) が $(\Gamma,\ \mathsf{t}_1\ \mathsf{t}_2)$ の解であるという仮定から、型付け関係の逆転補題より、ある T_1 が存在し、$\sigma\Gamma \vdash \sigma\mathsf{t}_1 : \mathsf{T}_1 \to \mathsf{T}$ かつ $\sigma\Gamma \vdash \sigma\mathsf{t}_2 : \mathsf{T}_1$ が得られる。帰納法の仮定より、$(\Gamma, \mathsf{t}_1, \mathsf{S}_1, C_1)$ の解 $(\sigma_1, \mathsf{T}_1 \to \mathsf{T})$ と、$(\Gamma, \mathsf{t}_2, \mathsf{S}_2, C_2)$ の解 (σ_2, T_1) が存在し、$\sigma_1 \backslash \mathcal{X}_1 = \sigma = \sigma_2 \backslash \mathcal{X}_2$ である。以下を満たす代入 σ' が存在することを示さなければならない。(1) $\sigma' \backslash \mathcal{X}$ と σ が一致すること、(2) $\sigma'\mathsf{X} = \mathsf{T}$ 、(3) σ' が C_1 と C_2 を単一化すること、(4) σ' が $\{\mathsf{S}_1 = \mathsf{S}_2 \to \mathsf{X}\}$ を単一化すること、つまり $\sigma'\mathsf{S}_1 = \sigma'(\mathsf{S}_2 \to \mathsf{X})$ 。σ' を以下のように定義する。

$$\sigma' = \left[\begin{array}{ll} \mathsf{Y} \mapsto \mathsf{U} & \mathsf{Y} \notin \mathcal{X} \text{ かつ } (\mathsf{Y} \mapsto \mathsf{U}) \in \sigma \text{ の場合} \\ \mathsf{Y}_1 \mapsto \mathsf{U}_1 & \mathsf{Y}_1 \in \mathcal{X}_1 \text{ かつ } (\mathsf{Y}_1 \mapsto \mathsf{U}_1) \in \sigma_1 \text{ の場合} \\ \mathsf{Y}_2 \mapsto \mathsf{U}_2 & \mathsf{Y}_2 \in \mathcal{X}_2 \text{ かつ } (\mathsf{Y}_2 \mapsto \mathsf{U}_2) \in \sigma_2 \text{ の場合} \\ \mathsf{X} \mapsto \mathsf{T} & \end{array}\right]$$

条件 (1) と (2) は自明に満たされる。$\mathcal{X}_1$ と $\mathcal{X}_2$ が重なりを持たないことから、(3) も満たされる。(4) を検査するためには、まずフレッシュさを保証する付帯条件に注目しよう。$FV(\mathsf{S}_1) \cap (\mathcal{X}_2 \cup \{\mathsf{X}\}) = \emptyset$ であるから、$\sigma'\mathsf{S}_1 = \sigma_1\mathsf{S}_1$ である。同様に $\sigma'\mathsf{S}_2 = \sigma_1\mathsf{S}_2$ も導ける。そして以下のように計算する。$\sigma'\mathsf{S}_1 = \sigma_1\mathsf{S}_1 = \mathsf{T}_1 \to \mathsf{T} = \sigma_2\mathsf{S}_2 \to \mathsf{T} = \sigma'\mathsf{S}_2 \to \sigma'\mathsf{X} = \sigma'(\mathsf{S}_2 \to \mathsf{X})$。

その他の場合

同様に示せる。 □

系 22.3.8. $\Gamma \vdash \mathsf{t} : \mathsf{S} \mid C$ とする。(Γ, t) の解が存在するとき、かつそのときに限り、$(\Gamma, \mathsf{t}, \mathsf{S}, C)$ の解が存在する。 □

［証明］ 定理 22.3.5 と定理 22.3.7 による。 □

演習 22.3.9. ［推奨, ★★★］ 製品レベルのコンパイラでは、規則 CT-App におけるフレッシュな型変数名の非決定的な選択は、呼ばれるたびに新しい型変数（これまでに生成したどの型変数とも異なり、検査中の文脈や項で明示的に言及されている型変数とも異なるもの）を生成する**関数**の呼び出しに置き換えられるのが典型的である。**“gensym”** と呼ばれるこの手の大域的な操作は、隠されたグローバル変数に対して副作用を持つため、形式的に論証するのが難しい。しかし、その振る舞いをかなり正確かつ数学的に扱いやすい方法で模倣することはできる。制約生成規則を通して、使われていない型変数名を「直列的に受け渡し」する方法である。

F は異なる型変数名の列を表すものとする。このとき、制約生成関係を $\Gamma \vdash \mathsf{t} : \mathsf{T} \ |_{\mathcal{X}}\ C$ と書く代わりに、$\Gamma \vdash_F \mathsf{t} : \mathsf{T} \ |_{F'}\ C$ と書く。Γ, F, t はアルゴリズムの入力であり、T, F', C は出力である。フレッシュな型変数が必要なときは、このアルゴリズムは F の最初の変数を取り出し、F の残りを F' として返す。

このアルゴリズムの規則を書き下せ。その規則と元の制約生成規則が、適切な意味で等価になることを証明せよ。 □

演習 22.3.10. ［推奨, ★★］ 演習 22.3.9 のアルゴリズムを ML で実装せよ。型には以下のデータ型を用いよ。

```
type ty =
    TyBool
  | TyNat
  | TyArr of ty * ty
  | TyId of string
```

制約集合には以下を用いよ。

```
type constr = (ty * ty) list
```

フレッシュな型変数名の無限列に対する表現も必要であろう。これを行う方法は幾つもある。以下は再帰データ型を用いたかなり直接的な方法である。

```
type nextuvar = NextUVar of string * uvargenerator
and uvargenerator = unit → nextuvar

let uvargen =
  let rec f n () = NextUVar("?X_" ^ string_of_int n, f (n+1))
  in f 0
```

つまり uvargen は、引数 () で呼ばれると、NextUVar(x,f) の形の値を返す関数である。x はフレッシュな型変数名であり、f は同じ形の別の関数である。 □

演習 22.3.11. [★★] 一般的な再帰関数定義（11.11 節）を扱うように制約生成アルゴリズムを拡張する方法を示せ。 □

22.4 単一化

制約集合の解を計算するには、Hindley [207] と Milner [307] によるアイディアを用いる。**単一化**（Robinson [423]）を用いて解の集合が空でないことを検査し、もしそうであればその「最良」の要素を見つける方法である。「最良」とは、その解から他の解を直接生成できるという意味である。

定義 22.4.1. 代入 σ と σ' について、$\sigma' = \gamma \circ \sigma$ となるような代入 γ が存在するとき、σ は σ' より**限定的でない**（または**より一般的である**）といい、$\sigma \sqsubseteq \sigma'$ と書く。 □

定義 22.4.2. 制約集合 C に対する**主要単一化子**（あるいは**最汎単一化子**）とは、C を充足し、また C を充足する任意の代入 σ' に対して $\sigma \sqsubseteq \sigma'$ となるような代入 σ を指す[†2]。 □

演習 22.4.3. [★] 以下の制約集合に対する主要単一化子を（存在するならば）書き下せ。

{X = Nat, Y = X→X}	{Nat→Nat = X→Y}
{X→Y = Y→Z, Z = U→W}	{Nat = Nat→Y}
{Y = Nat→Y}	{} （空の制約集合）

□

定義 22.4.4. 型の**単一化アルゴリズム**を図 22-2 のように定義する[†3]。2 行目の「let {S = T} ∪ C' = C」というところは、「集合 C から制約 S = T を選び、C の他の制約を C' として表す」という意味である。 □

[†2] (訳注) また、C を充足する任意の代入を単に**単一化子**と呼ぶ。

[†3] 実はこのアルゴリズムのどの部分も、単一化の対象が型式であることには依存していない。同じアルゴリズムを用いて、他の任意の種類の式（ただし一階のもの）に関する等式制約を解ける。

$$
\begin{array}{lll}
\mathit{unify}(C) & = & \text{if } C = \emptyset \text{ then } [\,] \\
& & \text{else let } \{\mathsf{S} = \mathsf{T}\} \cup C' = C \text{ in} \\
& & \quad \text{if } \mathsf{S} = \mathsf{T} \\
& & \quad\quad \text{then } \mathit{unify}(C') \\
& & \quad \text{else if } \mathsf{S} = \mathsf{X} \text{ and } \mathsf{X} \notin \mathit{FV}(\mathsf{T}) \\
& & \quad\quad \text{then } \mathit{unify}([\mathsf{X} \mapsto \mathsf{T}]C') \circ [\mathsf{X} \mapsto \mathsf{T}] \\
& & \quad \text{else if } \mathsf{T} = \mathsf{X} \text{ and } \mathsf{X} \notin \mathit{FV}(\mathsf{S}) \\
& & \quad\quad \text{then } \mathit{unify}([\mathsf{X} \mapsto \mathsf{S}]C') \circ [\mathsf{X} \mapsto \mathsf{S}] \\
& & \quad \text{else if } \mathsf{S} = \mathsf{S}_1 \to \mathsf{S}_2 \text{ and } \mathsf{T} = \mathsf{T}_1 \to \mathsf{T}_2 \\
& & \quad\quad \text{then } \mathit{unify}(C' \cup \{\mathsf{S}_1 = \mathsf{T}_1, \mathsf{S}_2 = \mathsf{T}_2\}) \\
& & \quad \text{else} \\
& & \quad\quad \mathit{fail}
\end{array}
$$

▲ 図 22-2. 単一化アルゴリズム[†4]

5 行目の付帯条件 $\mathsf{X} \notin FV(\mathsf{T})$ と 7 行目の $\mathsf{X} \notin FV(\mathsf{S})$ は、**出現検査**として知られる。これは、アルゴリズムが $\mathsf{X} \mapsto \mathsf{X}\to\mathsf{X}$ のような循環的な代入を含む解を生成しないようにする効果がある。そのような代入は有限の型式について議論している場合には意味を成さない。(もし**無限**の型式、つまり第 20 章や第 21 章にあるような再帰型を含むように言語を拡張するならば、出現検査は省略できる。)

定理 22.4.5. アルゴリズム *unify* は常に停止する。単一化できない制約集合を入力として与えられたときは失敗し、そうでないときは主要単一化子を返す。より形式的には、

(1) $\mathit{unify}(C)$ はすべての C に対して、失敗するか代入を返すことで停止する。
(2) $\mathit{unify}(C) = \sigma$ ならば、σ は C の単一化子である。
(3) δ が C の単一化子ならば、$\mathit{unify}(C) = \sigma$ かつ $\sigma \sqsubseteq \delta$ である。 □

[証明] (1) について。制約集合 C の**次数**を組 (m, n) として定義する。m は C における異なる型変数の数である。n は C の型の合計サイズである。*unify* アルゴリズムの各節は、直ちに停止する(最初の場合では成功、最後の場合では失敗)か、より辞書的に小さい次数の制約集合で *unify* を再帰的に呼び出す。これは容易に確かめられる。

(2) は $\mathit{unify}(C)$ の計算における再帰呼び出しの回数に関する単純な帰納法による。変数がかかわる二つの場合以外は自明である。これらの二つの場合は、次の観察に基づいて証明できる。任意の制約集合 D に対して、σ が $[\mathsf{X} \mapsto \mathsf{T}]D$ を単一化するなら $\sigma \circ [\mathsf{X} \mapsto \mathsf{T}]$ は $\{\mathsf{X} = \mathsf{T}\} \cup D$ を単一化する。

(3) も $\mathit{unify}(C)$ の計算における再帰呼び出しの回数に関する帰納法からいえる。C が空であれば、$\mathit{unify}(C)$ は直ちに自明な代入 $[\,]$ を返す。$\delta = \delta \circ [\,]$ であるから、所要の通り $[\,] \sqsubseteq \delta$ を得る。C が空でなければ、$\mathit{unify}(C)$ は C から適当な等式 $\mathsf{S} = \mathsf{T}$ を選び、S と T の形に基づいて場合分けを行う。

$\mathsf{S} = \mathsf{T}$ の場合

δ は C の単一化子であるから、C' も単一化する。帰納法の仮定により、所要の通り、$\mathit{unify}(C) = \sigma$ かつ $\sigma \sqsubseteq \delta$ である。

$\mathsf{S} = \mathsf{X}$ かつ $\mathsf{X} \notin FV(\mathsf{T})$ の場合

δ は S と T を単一化するから、$\delta(\mathsf{X}) = \delta(\mathsf{T})$ を得る。よって任意の型 U に対して、$\delta(\mathsf{U}) = \delta([\mathsf{X} \mapsto$

[†4] (訳注) 制約集合に対する代入 σC は、σ を各要素に対して適用した $\{\sigma\mathsf{T} = \sigma\mathsf{S} \mid (\mathsf{T} = \mathsf{S}) \in C\}$ を表す。

T]U) を得る。特に δ は C' を単一化するから、$[\mathsf{X} \mapsto \mathsf{T}]C'$ も単一化することになる。帰納法の仮定により、$\mathit{unify}([\mathsf{X} \mapsto \mathsf{T}]C') = \sigma'$ であり、$\delta = \gamma \circ \sigma'$ となる γ が存在することがわかる。$\mathit{unify}(C) = \sigma' \circ [\mathsf{X} \mapsto \mathsf{T}]$ であるから、$\delta = \gamma \circ (\sigma' \circ [\mathsf{X} \mapsto \mathsf{T}])$ を示すことで議論が完了する。よって任意の型変数 Y を考える。もし $\mathsf{Y} \neq \mathsf{X}$ であれば、明らかに $(\gamma \circ (\sigma' \circ [\mathsf{X} \mapsto \mathsf{T}]))\mathsf{Y} = (\gamma \circ \sigma')\mathsf{Y} = \delta\mathsf{Y}$ である。一方、上で示した通り $(\gamma \circ (\sigma' \circ [\mathsf{X} \mapsto \mathsf{T}]))\mathsf{X} = (\gamma \circ \sigma')\mathsf{T} = \delta\mathsf{X}$ である。以上の観察の組み合わせにより、任意の変数 Y に対して $\delta\mathsf{Y} = (\gamma \circ (\sigma' \circ [\mathsf{X} \mapsto \mathsf{T}]))\mathsf{Y}$ であるとわかる。つまり $\delta = (\gamma \circ (\sigma' \circ [\mathsf{X} \mapsto \mathsf{T}]))$ である。

$\mathsf{T} = \mathsf{X}$ かつ $\mathsf{X} \notin FV(\mathsf{S})$ の場合
同様。

$\mathsf{S} = \mathsf{S}_1 \to \mathsf{S}_2$ かつ $\mathsf{T} = \mathsf{T}_1 \to \mathsf{T}_2$ の場合
そのまま。δ が $\{\mathsf{S}_1 \to \mathsf{S}_2 = \mathsf{T}_1 \to \mathsf{T}_2\} \cup C'$ の単一化子であるとき、かつそのときに限り、δ は $C' \cup \{\mathsf{S}_1 = \mathsf{T}_1, \mathsf{S}_2 = \mathsf{T}_2\}$ の単一化子であることに注目すればよい。

S と T が上記のいずれの場合にも当てはまらない場合は、*unify*(C) は失敗する。しかしこれは二通りの場合でしか起こりえない。S が Nat で T が矢印型（もしくはその反対）である場合か、$\mathsf{S} = \mathsf{X}$ かつ $\mathsf{X} \in \mathsf{T}$（もしくはその反対）の場合だけである。前者の場合は明らかに C が単一化可能という仮定と矛盾している。後者の場合も矛盾することを確かめるには、仮定により $\delta\mathsf{S} = \delta\mathsf{T}$ であることを思い出せばよい。もし X が T の中に現れる場合、$\delta\mathsf{T}$ は常に $\delta\mathsf{S}$ より真に大きいことになる。したがって、*unify*(C) が失敗するとき、C は単一化可能でない。これは δ が C の単一化子であるという仮定と矛盾する。よってこの場合も起こりえない。 □

演習 22.4.6. [推奨, ★★★] 単一化アルゴリズムを実装せよ。 □

22.5 主要型

型変数を具体化して項を型付け可能にする方法が**何かしら**存在するなら、それを行う**最も一般的**な、つまり**主要**な方法があるということを前述した。ここで、この観察を形式化する。

定義 22.5.1. $(\Gamma, \mathsf{t}, \mathsf{S}, C)$ の**主要解**とは、解 (σ, T) であり、(σ', T') が $(\Gamma, \mathsf{t}, \mathsf{S}, C)$ の解であるならば常に $\sigma \sqsubseteq \sigma'$ であるようなものを指す。(σ, T) が主要解のとき、T を Γ の下での t の**主要型**と呼ぶ[†5]。 □

演習 22.5.2. [★ ↛] 空の型付け文脈の下での、`λx:X. λy:Y. λz:Z. (x z) (y z)` の主要型は何か。 □

定理 22.5.3. [**主要型**] $(\Gamma, \mathsf{t}, \mathsf{S}, C)$ が解を持つならば、主要解が存在する。図 22-2 の単一化アルゴリズムを使うと $(\Gamma, \mathsf{t}, \mathsf{S}, C)$ が解を持つかどうかを決定でき、また、解を持つ場合は主要解を計算できる。 □

[証明] $(\Gamma, \mathsf{t}, \mathsf{S}, C)$ の解の定義と、単一化の性質による。 □

系 22.5.4. (Γ, t) が解を持つかどうかは決定可能である。 □

[証明] 系 22.3.8 と定理 22.5.3 による。 □

---
†5 主要型と**主要型付け**を混同しないこと。264 ページを見よ。

演習 22.5.5. [推奨, ★★★ ↛] 演習 22.3.10 の制約生成と演習 22.4.6 の単一化アルゴリズムを組み合わせて、主要型を計算する型検査器を構成せよ。ここでは検査器 reconbase を出発点にせよ。型検査器との典型的なやり取りは以下のようになるだろう。

```
  λx:X. x;
▸ <fun> : X → X

  λz:ZZ. λy:YY. z (y true);
▸ <fun> : (?X₀→?X₁) → (Bool→?X₀) → ?X₁

  λw:W. if true then false else w false;
▸ <fun> : (Bool→Bool) → Bool
```

$?X_0$ のような名前を持つ型変数は自動的に生成されている。 □

演習 22.5.6. [★★★] 上記の定義（定義 22.3.2 など）を、レコードが扱えるように拡張する場合、どのような問題が発生するだろうか。それはどのように解決できるだろうか。 □

主要型のアイディアを用いると、ここまで我々が説明してきたアルゴリズムより段階的に動作する型再構築アルゴリズムを構成できる。制約をすべて最初に生成してから解こうとするのではなく、生成と解決を交互に行うことで、型再構築アルゴリズムが各段階で主要型を実際に返すようになる。型が常に主要型であるということは、アルゴリズムが部分項を解析し直す必要がないということを保証する。各段階で、型付け可能性を達成するために必要な最小限の約束しかしないということである。そのようなアルゴリズムの主な利点は、ユーザのプログラムの誤りをより正確に指摘できることである。

演習 22.5.7. [★★★ ↛] 演習 22.5.5 の解答を変更して、段階的に単一化を行い、主要型を返すようにせよ。 □

22.6 暗黙の型注釈

型再構築を備える言語では通常、プログラマはラムダ抽象の型注釈を完全に省略できる。これを達成する方法として、(22.2 節で述べたように）単純に、型注釈が省略されているところへフレッシュに生成した型変数を埋め込む処理を構文解析器にやらせる方法がある。もっと良い方法は別にある。型注釈が省略された抽象を項の構文に追加し、さらに対応する規則を制約型付け関係に追加する方法である。

$$\frac{\mathtt{X} \notin \mathcal{X} \qquad \Gamma, \mathtt{x}{:}\mathtt{X} \vdash \mathtt{t}_1 : \mathtt{T} \ |_{\mathcal{X}}\ C}{\Gamma \vdash \lambda\mathtt{x}.\mathtt{t}_1 : \mathtt{X}\to\mathtt{T} \ |_{\mathcal{X}\cup\{\mathtt{X}\}}\ C} \qquad \text{(CT-AbsInf)}$$

型注釈のない抽象をこのように扱うことは、糖衣構文とみなすよりもわずかに直接的である。また、少しだが便利な形で表現力も増える。型注釈のない抽象の**コピー**を複数作るとき、CT-AbsInf 規則によって各コピーの引数の型に**異なる**変数を選べる。一方、型注釈のない抽象を見えない型変数で注釈されているとみなす場合は、コピーは**同じ**引数の型を共有する複数の式となる。この違いは次節の let 多相の議論で重要になってくる。

22.7 let 多相

多相性という言葉は、プログラムの一部分を、異なる文脈では異なる型として用いることを可能にする各種の言語機構を指す（23.2 節で数種類の多相性について詳しく議論する）。上記で示した型再構築アルゴリズムを一般化すると、**let 多相**（**ML 式**多相、**Damas-Milner** 多相とも）として知られる一種の単純な多相性が得られる。この機能は ML の初版（Milner [307]）で導入され、成功した言語設計の数々に組み込まれてきた。この機能は、リスト、配列、ツリー、ハッシュテーブル、ストリーム、ユーザインターフェイスウィジェットなど、一般的な構造を扱う強力な**総称ライブラリ**の基礎をなす。

let 多相のきっかけは、以下のような例である。double という簡単な関数を定義して使用することを考える。この関数は第二引数に対して第一引数を連続して二回適用する。

```
let double = λf:Nat→Nat. λa:Nat. f(f(a)) in
double (λx:Nat. succ (succ x)) 2;
```

型 Nat→Nat の関数に double を適用したいので、double には型 (Nat→Nat)→(Nat→Nat) を与える型注釈を付ける。その一方で、ブール値関数を二回適用するためにも使用できるように double を定義することもできる。

```
let double = λf:Bool→Bool. λa:Bool. f(f(a)) in
double (λx:Bool. x) false;
```

しかし、ブール値と数の**両方**に対して同じ double 関数を用いることはできない。もし同じプログラムの中で両方必要な場合は、型注釈以外は全く同じ関数を二つ定義しなければならない。

```
let doubleNat = λf:Nat→Nat. λa:Nat. f(f(a)) in
let doubleBool = λf:Bool→Bool. λa:Bool. f(f(a)) in
let a = doubleNat (λx:Nat. succ (succ x)) 1 in
let b = doubleBool (λx:Bool. x) false in ...
```

double の抽象に**型変数**で注釈したとしても無駄である。

```
let double = λf:X→X. λa:X. f(f(a)) in ...
```

例えば以下のように書くと、

```
let double = λf:X→X. λa:X. f(f(a)) in
let a = double (λx:Nat. succ (succ x)) 1 in
let b = double (λx:Bool. x) false in ...
```

a の定義で double を用いているので制約 X→X = Nat→Nat が生成されるが、一方で b の定義で double を用いているので制約 X→X = Bool→Bool が生成される。これらの制約は X に関して充足不可能な要求を行っているので、プログラム全体が型付け不可能になる。

何がまずかったのだろうか。この例では、変数 X が異なる二つの役割を持つ。一つ目は、a の計算において、double の第一引数が関数であり、その定義域と値域の型が共に double の第二引数の型（Nat）と同じでなければならないという制約を表現することである。二つ目は、b の計算においても、double の引数が同様に関連付けられなければならないことである。残念ながら、同じ変数 X が両方の場合で用いられるため、二か所にある double の第二引数が同じ型を持たなければならない、という誤った制約ができてしまう。

やりたいのは、最後のつながりを断つことである。つまり、double が使われるたびにそれぞれに**異なる**変数 X を関連付けたい。幸い、これは簡単に達成できる。まずは let の通常の型付けを変更する。これまでは右辺 t_1 の型を計算し、その型を本体 t_2 の型を計算する際に束縛変数 x の型とし

ていたが、

$$\frac{\Gamma \vdash \mathtt{t}_1 : \mathtt{T}_1 \qquad \Gamma, \mathtt{x}{:}\mathtt{T}_1 \vdash \mathtt{t}_2 : \mathtt{T}_2}{\Gamma \vdash \mathtt{let}\ \mathtt{x}{=}\mathtt{t}_1\ \mathtt{in}\ \mathtt{t}_2 : \mathtt{T}_2} \tag{T-Let}$$

そうではなく、本体中の x に $\mathtt{t}_1$ を**代入**し、展開された式の型検査を行うようにする。

$$\frac{\Gamma \vdash [\mathtt{x} \mapsto \mathtt{t}_1]\mathtt{t}_2 : \mathtt{T}_2}{\Gamma \vdash \mathtt{let}\ \mathtt{x}{=}\mathtt{t}_1\ \mathtt{in}\ \mathtt{t}_2 : \mathtt{T}_2} \tag{T-LetPoly}$$

let の制約型付け規則も同様に書く。

$$\frac{\Gamma \vdash [\mathtt{x} \mapsto \mathtt{t}_1]\mathtt{t}_2 : \mathtt{T}_2 \ |_{\mathcal{X}}\ C}{\Gamma \vdash \mathtt{let}\ \mathtt{x}{=}\mathtt{t}_1\ \mathtt{in}\ \mathtt{t}_2 : \mathtt{T}_2 \ |_{\mathcal{X}}\ C} \tag{CT-LetPoly}$$

要するに、let の型付け規則を変えて、型を計算する前に評価を 1 ステップ行うようにした。

$$\mathtt{let}\ \mathtt{x}{=}\mathtt{v}_1\ \mathtt{in}\ \mathtt{t}_2 \longrightarrow [\mathtt{x} \mapsto \mathtt{v}_1]\mathtt{t}_2 \tag{E-LetV}$$

それから double の定義を、22.6 節の**暗に注釈された**ラムダ抽象を使って書き換える。

```
let double = λf. λa. f(f(a)) in
let a = double (λx:Nat. succ (succ x)) 1 in
let b = double (λx:Bool. x) false in ...
```

let の制約型付け規則（CT-LetPoly）と暗に注釈されたラムダ抽象の規則（CT-AbsInf）を組み合わせると、まさに必要なものが得られる。CT-LetPoly は double の定義のコピーを二つ作り、CT-AbsInf は各抽象に対して異なる型変数を割り当てる。あとは従来の制約解決と同じ手順である。

しかしこのやり方には、実際に使えるようになる前に解決しないといけない問題が幾つかある。わかりやすい問題としては、let で束縛された変数を let の本体の中で使っていない場合、その定義が型検査されない。例えば以下のようなプログラムで型検査が成功する。

```
let x = <全く壊れたコード> in 5
```

これを修正するには、型付け規則に前提を以下のように追加し、

$$\frac{\Gamma \vdash [\mathtt{x} \mapsto \mathtt{t}_1]\mathtt{t}_2 : \mathtt{T}_2 \qquad \Gamma \vdash \mathtt{t}_1 : \mathtt{T}_1}{\Gamma \vdash \mathtt{let}\ \mathtt{x}{=}\mathtt{t}_1\ \mathtt{in}\ \mathtt{t}_2 : \mathtt{T}_2} \tag{T-LetPoly}$$

さらに CT-LetPoly にも相応の前提を追加することで、$\mathtt{t}_1$ が正しく型付けされることを保証すればよい。

関連する問題として、let の本体の中で let の束縛変数が**何回も**出現する場合、let 定義の右辺全体が出現ごとに検査される。これは暗に注釈されたラムダ抽象を含むか含まないかによらない。右辺自身も let 束縛を含むことができるので、この型付け規則のせいで、型検査器は元の項のサイズに対して指数的な量の計算をしなければならないことになりえてしまう。

このような型検査のやり直しを避けるために、let 多相を持つ言語の実用的な実装では、実際のところ、型付け規則を（形式的に等価ではあるが）より巧妙に定式化し直したものを用いる。概略としては、文脈 Γ における項 $\mathtt{let}\ \mathtt{x}{=}\mathtt{t}_1\ \mathtt{in}\ \mathtt{t}_2$ の型検査を以下のように行う。

(1) 制約型付け規則を使って、右辺 $\mathtt{t}_1$ に対して型 $\mathtt{S}_1$ と関連する制約の集合 C_1 を計算する。

(2) 単一化を用いて制約集合 C_1 に対する最も一般的な解 σ を見つけ、σ を $\mathtt{S}_1$ に適用し、$\mathtt{t}_1$ の**主要型** $\mathtt{T}_1$ を得る。

(3) T_1 に残っている任意の型変数を**一般化**する。$X_1 \ldots X_n$ が残っている型変数だとすると、t_1 の主要**型スキーム**を $\forall X_1 \ldots X_n . T_1$ と書く。
ここで一つ注意が必要である。T_1 だけでなく Γ でも言及されている変数は一般化**しない**よう気をつけないといけない。これらの変数は、t_1 とその文脈の間の制約として実際に必要なものに対応しているためである。例えば、

```
λf:X→X. λx:X. let g=f in g(x);
```

の中で、g の型 X→X の型変数 X を一般化すべきでない。一般化してしまうと、以下のような誤ったプログラムを型付けできてしまうことになる。

```
(λf:X→X. λx:X. let g=f in g(0))
  (λx:Bool. if x then true else false)
  true;
```

(4) 束縛変数 x に対して型スキーム $\forall X_1 \ldots X_n . T_1$ を記録するよう文脈を拡張してから、本体 t_2 の型検査を始める。一般的に、文脈はこのあと各自由変数に対して単なる型だけでなく、型スキームを対応付けるようになる。

(5) t_2 の中で x が現れるたびに、その型スキーム $\forall X_1 \ldots X_n . T_1$ を引いてくる。そしてフレッシュな型変数 $Y_1 \ldots Y_n$ を生成し、それらを用いて型スキームを**具体化**する。これにより $[X_1 \mapsto Y_1, \ldots, X_n \mapsto Y_n]T_1$ が得られる。これを x の型として用いる[†6]。

このアルゴリズムは、型検査前に let 式を代入によって消してしまう極めて単純な方法に比べるとずっと効率的である。実際、実用上は入力プログラムのサイズに対して「基本的に線形」に見えることが数十年間の経験によってわかっている。それだけに、最悪計算量はやはり指数的であると Kfoury, Tiuryn, and Urzyczyn [258] と Mairson [298] が独立して示したときは、大変な驚きであった。彼らが構成した例は、深く入れ子になった let の列を用いる。この例で構成される式の型は、式自体に対して指数的に大きくなる。例えば、（Mairson [298] による）以下の OCaml プログラムは、正しく型付けできるものの、型検査に非常に長い時間がかかってしまう。

```
let f0 = fun x → (x,x) in
  let f1 = fun y → f0(f0 y) in
    let f2 = fun y → f1(f1 y) in
      let f3 = fun y → f2(f2 y) in
        let f4 = fun y → f3(f3 y) in
          let f5 = fun y → f4(f4 y) in
            f5 (fun z → z)
```

その理由を見るには、f_0、f_1、… というように、OCaml のトップレベルに一つずつ入力していくとよい[†7]。より詳しい議論は Kfoury, Tiuryn, and Urzyczyn [261] を見よ。

最後に話しておくべきことは、let 多相を持つ本格的なプログラミング言語の設計においては、書

[†6] 型変数で明示的に注釈されたラムダ抽象と、制約生成アルゴリズムが変数を生成する注釈のない抽象の違いは、一般化と具体化を導入した時点で意味をなさなくなる。どちらにしても、let の右辺には変数を含む型が割り当てられる。その変数は文脈に追加される前に一般化され、具体化されるたびにフレッシュな変数に置き換えられる。

[†7] （訳注） 現在の OCaml の実装において、このプログラムの型検査に時間がかかる実際の原因は、出現検査の実装と型の表示にある。出現検査の実装や型の表示が原因ではない、OCaml で型検査に時間のかかる別の例としては、

```
let l0 = ([], [], [], []) in
let l1 = (l0, l0, l0, l0) in
let l2 = (l1, l1, l1, l1) in
... (中略) ...
let l9 = (l8, l8, l8, l8) in
ignore (l9)
```

のようなプログラムがある（この例はジャック・ガリグ氏による）。

き換え可能な記憶域セルのように副作用のある機能と多相性との間の相互作用に少し注意を要するということである。その危険性は以下の例で見ることができる。

```
let r = ref (λx. x) in
(r:=(λx:Nat. succ x);
 (!r)true);
```

上記で概説したアルゴリズムを使うと、`let` の右辺の主要型は `Ref(X→X)` になる。`X` は他のどこにも現れないため、この型は `∀X.Ref(X→X)` に一般化される。文脈に追加するとき `r` に割り当てられるのは、この型スキームとなる。この型スキームは、2 行目の破壊的代入を型検査するとき、`Ref(Nat→Nat)` に具体化される。しかし 3 行目を型検査するときは、`Ref(Bool→Bool)` に具体化される。これは健全でない。この項が評価されたとき、`succ` を `true` に適用するという結果になってしまうためである。

この問題は、型付け規則が評価規則と同期していないということである。この節で導入された型付け規則は、`let` 式に出くわしたらその右辺を本体に**直ちに**代入すべきである、といっている。しかし評価規則は、右辺が値に簡約された**後に**のみ代入できる、といっている。型付け規則は `ref` 構築子が二か所で使われているのを見て、それらを異なる仮定の下で解析するが、実行時に実際に確保される `ref` は一つしかない。

この不整合を修正するには、評価か型付けのどちらかを変えればよい。前者の場合、`let` の評価規則は以下のようになる[†8]。

$$\texttt{let x=t}_1\texttt{ in t}_2 \longrightarrow [\texttt{x} \mapsto \texttt{t}_1]\texttt{t}_2 \qquad \text{(E-LET)}$$

この評価戦略の下では、上記の危険な例を評価する最初のステップで、`r` がその定義に置き換えられる。その結果、以下のようになる。

```
(ref (λx. x)) := (λx:Nat. succ x);
(!(ref (λx. x))) true;
```

これは文句なく安全である。1 行目で恒等関数で初期化された参照セルを作り、それに `(λx:Nat. succ x)` を保存する。2 行目では、恒等関数を持つ**別の**参照を作り、その中身を取り出して `true` に適用する。しかしこの計算が示しているのは、評価規則を型付け規則に合わせて変更すると、もはや値呼び評価順序の普通の直観にはそぐわない、やや奇妙な意味論を持つ言語になってしまうということである。(値呼びでない評価戦略を持つ手続き型の言語がないわけではない (Augustsson [29])。しかし実行時の副作用の順序が理解も制御も難しく、一般的にはなっていない。)

評価規則に合わせて型付け規則を変更するほうがよい。幸い、これは簡単である。右辺が構文的に値であるときのみ、`let` 束縛を多相的に扱える (つまり自由な型変数を一般化できる) という制限を課すだけでよい (この制限はしばしば**値制限**と呼ばれる)。これはつまり、上記の危険な例において、文脈に追加するとき `r` に割り当てられる型が `∀X.Ref(X→X)` ではなく `Ref(X→X)` になるということである。2 行目によって課される制約により、`X` は `Nat` に強制される。`Nat` は `Bool` と単一化できないので、3 行目の型検査で失敗することになる。

値制限は型安全性に関する問題を解決するが、表現力が少し犠牲になる。`let` 式の右辺が何か面白い計算をするが多相型スキームを割り当てても問題ない、というようなプログラムを書けなくなってしまう。驚くべきことに、この制限は実用上ほとんど問題にならない。このことは Wright

---

[†8] 厳密に言えば、参照を持つ言語について議論しているのだから、第 13 章で行ったように規則にストアを注釈すべきである。

$$\texttt{let x=t}_1\texttt{ in t}_2 \mid \mu \longrightarrow [\texttt{x} \mapsto \texttt{t}_1]\texttt{t}_2 \mid \mu \qquad \text{(E-LET)}$$

[499] が決着をつけた。Wright は、**弱型変数**に基づく、より柔軟な `let` 型付け規則を提供する ML 方言（Standard ML の 1990 年の定義（Milner, Tofte, and Harper [315]））で書かれたコードの巨大なコーパスを解析し、（ごく少数を除いて）右辺はすべて構文的な値であるということに気づいた。この観察結果によって、議論はおおよそ決着した。今では、ML 式の let 多相を持つ主な言語はすべて値制限を採用している。

演習 22.7.1. [★★★ ↛] この節で概説したアルゴリズムを実装せよ。 □

22.8 注記

ラムダ計算に対する主要型の概念は、少なくとも 1950 年代の Curry の研究成果（Curry and Feys [128]）まで遡る。Curry のアイディアに基づいた主要型の計算アルゴリズムは Hindley [207] が与えた。同様のアルゴリズムは、独立して Morris [326] と Milner [307] も開発した。命題論理の世界ではこのアイディアはさらに遡る。おそらく 1920 年代の Tarski まで、少なくとも 1950 年代の Meredith 従兄弟（Lemmon, Meredith, Meredith, Prior, and Thomas [278]）までは確実に遡れる。計算機上で動く最初の実装は David Meredith が 1957 年に作った。主要型に関するさらなる歴史的論評は Hindley [209] に見られる。

単一化（Robinson [423]）は計算機科学の多くの分野の基礎となっている。詳細な入門書は、例えば Baader and Nipkow [31]、Baader and Siekmann [32]、Lassez and Plotkin [271] がある。

ML 式の let 多相に最初に言及したのは Milner [307] である。型再構築アルゴリズムは数多く提案されている。特筆すべきは、Damas と Milner による古典的な **Algorithm W** である [129]（Lee and Yi [276] も見よ）。Algorithm W と本章で示したものとの大きな違いは、前者が「純粋型再構築」に特化している（完全な**型無し**ラムダ項に主要型を割り当てる）が、我々は型検査と型再構築を組み合わせ、明示的な型注釈を持つ項を許したことである（この注釈は型変数を含んでもよいが、その必要があるわけではない。このせいで、我々の手法の説明は少し複雑になっている（特に定理 22.3.7 の完全性の証明。プログラマが書いた型変数と制約生成規則が導入した型変数とを区別するよう注意しなければならない）が、他の章の形式と相性がよい。

Cardelli [75] による古典的な論文は、数多くの実装上の問題を明らかにした。型再構築アルゴリズムの他の提案は Appel [19]、Aho et al. [16]、Reade [396] に見られる。**mini-ML**（Clement, Despeyroux, Despeyroux, and Kahn [102]）と呼ばれる核となる体系の非常に洗練された説明は、しばしば理論的な議論の基礎をなす。Tiuryn [462] はさまざまな型再構築の問題をサーベイしている。

主要型と、似た概念である**主要型付け**とを混同してはならない。主要型を計算するとき、文脈 Γ と項 t はアルゴリズムの入力と考えられ、主要型 T はその出力である。主要型付けを計算するアルゴリズムは t だけを入力として受け取り、Γ と T を出力として返す。つまり t 中の自由変数の型について**最小の仮定**を計算する。主要型付けを用いると、分割コンパイルや "smartest recompilation"[†9] の機能を提供できたり、段階的な型推論を行えたり、型エラーを正確に示せたりする。残念ながら多くの言語（特に ML）は、主要型を持つが主要型付けは持たない。Jim [239] を参照せよ。

強力さと単純さの印象的な組み合わせを備える ML 式多相は、言語設計の空間の「芯」を捉えている。他の洗練された型付け機能と組み合わせるには、細心の注意が必要な場合が多いことが知られている。この分野で最も成功したのは、レコード型のための型再構築を明確に説明したことで

[†9] （訳注）Shao, Zhong and Andrew W. Appel. Smartest recompilation. *20th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, pages 439–450, 1993.

ある。これは Wand [487] が提案し、Wand 自身 [488; 490] や Rémy [399; 400; 401; 402; 405] 他多くの人々が発展させた。アイディアとしては、**列変数** と呼ばれる新しい種類の変数を導入する。この変数は型の上ではなく、フィールドラベルと関連する型をまとめた「列」の上を動く。列変数を含む制約集合を解くには、ある種の単純な**等式単一化**を用いる。演習 22.5.6 を見よ。Garrigue [169] などは、関連する手法をバリアント型に対して開発した。これらの技法は、**型クラス**（Kaes [246]; Wadler and Blott [484])、**制約型**（Odersky, Sulzmann, and Wehr [343])、**限定型**（Jones [242; 241]）の一般的な概念に拡張されてきた。これらは Haskell の**型クラス**システムの基礎をなしている（Hall et al. [190]; Hudak et al. [226]; Thompson [461])。同様のアイディアは Mercury（Somogyi, Henderson, and Conway [439]）や Clean（Plasmeijer [380]）にも現れる。

第 23 章で議論する、**非可述的多相**というより強力な形式に対する型再構築は決定不能であることが Wells [492] によって証明された。さらに、この体系は幾つかの**部分的型再構築**についても決定不能であると知られている。その結論や、**ランク 2 多相**のようなより強力な多相性と ML 式の型再構築を組み合わせる方式については、23.6 節と 23.8 節でもう少し触れる。

部分型付けと ML 式の型再構築を組み合わせることに関しては、有望な初期成果が報告されている（Aiken and Wimmers [17]; Eifrig, Smith, and Trifonov [144]; Jagannathan and Wright [236]; Trifonov and Smith [467]; Odersky, Sulzmann, and Wehr [343]; Flanagan and Felleisen [156]; Pottier [391]）が、実用的な検査器はまだ広く普及していない。

再帰型（第 20 章）を扱えるように ML 式の型再構築を拡張しても、特に問題は**起きない**ことが示されている（Huet [227; 228])。この拡張アルゴリズムが本章で説明したアルゴリズムとはっきり異なるのは、単一化の定義の中で**出現検査**を行わないという点だけだ（出現検査は通常、単一化アルゴリズムの返す代入が循環しないことを保証する)。この変更を加えると、停止性を保証するために単一化アルゴリズムで用いられる内部表現も変更して、共有を維持するようにする必要がある。例えば、（循環しうる）ポインタ構造に対する破壊的な操作を用いる必要がある。このような内部表現は高性能な実装では一般的である。

一方、型再構築を再帰的に定義された**項**と組み合わせると、**多相再帰**と呼ばれるややこしい問題が発生する。ML の再帰的な関数定義に対する単純な（しかし問題のない）型付け規則は、再帰関数が自分自身の定義本体で単相的に用いられる（つまり、引数や返り値の型がすべての再帰呼び出しにおいて一定でなければならない）一方で、プログラムの他の箇所の出現では多相的に用いられる（異なる型の引数・返り値を使える)、と規定している。Mycroft [331] と Meertens [306] は再帰的定義に対する多相型付け規則を提案した。この規則は、再帰関数の本体から自分自身を呼び出す再帰呼び出しを異なる型で具体化することを許している。この拡張はしばしば **Milner-Mycroft 計算**と呼ばれる。この拡張の再構築問題が決定不能であることは、Henglein [202] と Kfoury, Tiuryn, and Urzyczyn [259] が独立に示した。どちらの証明も（無制限の）半単一化問題の非決定性（これは Kfoury, Tiuryn, and Urzyczyn [260] が示した）に依存している。

第 23 章

全称型

本章では主に、純粋な System F を扱う（図 23-1）。ただし 23.4 節では、これまでに学んだ諸々の機能拡張を使っている。本章に対応する OCaml 実装は `fullpoly` である。（二つ組やリストを使う事例では `fullomega` 検査器が必要である。）

前章では、ML で使われる単純な **let 多相**について学んだ。本章では、**System F** という強力な計算体系の下で、より一般的な多相性について考える。

23.1 動機

22.7 節で述べたように、単純型付きラムダ計算での "**double**" 関数は無限にある。

```
doubleNat = λf:Nat→Nat. λx:Nat. f (f x);
doubleRcd = λf:{l:Bool}→{l:Bool}. λx:{l:Bool}. f (f x);
doubleFun = λf:(Nat→Nat)→(Nat→Nat). λx:Nat→Nat. f (f x);
```

これらの関数は異なる型の引数に適用可能であるが、挙動はどれも全く同じである（実際、型注釈のところ以外、プログラムの字面が完全に同じである）。もしこの `double` 演算を別の型の引数に適用したいなら、同じプログラムの中でも、`doubleT` の定義を T ごとに逐一書いていく必要がある。こういうカットアンドペーストによるプログラミングは、以下のソフトウェア工学の基本原則に反する。

> 抽象化原則：プログラムの機能の重要なところは、それぞれソースコード中の一か所だけで実装されるべきである。コード中の異なる箇所で似た機能が実現されているとき、差分を**抽象化**して一つにまとめることは、一般的に有益である。

今の場合、差分は型なのだ。よって必要なのは、項から型を抽象化する機能と、抽象化した項を具体的な型注釈で具体化する機能ということになる。

23.2 さまざまな多相性

一つのコードを複数の型で扱えるようにした型システムを、まとめて**多相的な**（polymorphic）型システムと呼ぶ（**poly** = many = 多、**morph** = form = 相）。現代的な言語の多相性には、幾つか種類がある（この分類は Strachey [446] および Cardelli and Wegner [88] に由来する）。

- 本章の主題は**パラメータ多相**である。この多相性では実際の型の代わりに変数を使って、一つのコードを「総称的に」型付けでき、それを必要に応じて特定の型に具体化できる。パラメータ化された定義は**一様**である。すなわち、その具体化はすべて同じ振る舞いをする。
- 最も強力な形のパラメータ多相は**非可述的多相**（**一級多相**とも）である。この章ではこの多相性を扱う。実用上は **ML 式多相**（**let 多相**とも）のほうが一般的である。こちらは、多相性をトップレベルの `let` 束縛だけに限定し、多相的な値を引数として受け取る関数を許さない。その代わり、便利で自然な形の自動的**型再構築**（第 22 章）が可能になる。一級多相もプログラミング言語に普及しつつあり、ML のような言語の強力なモジュールシステムの技術的基盤となっている（Harper and Stone [199] を参照されたい）。

- 対照的に、**アドホック多相**では、多相的な値が異なる型として「見られる」とき、異なったように振る舞うことを可能にする。アドホック多相の最も一般的な例は**オーバーロード**である。これは一つの関数名に複数の実装を割り当てる機能である。関数適用ごとに、その引数の型に基づいて、コンパイラ（または実行時システム、これはオーバーロードの解決が**静的**か**動的**かによる）が適切な実装を選ぶ。
- 関数のオーバーロードを一般化すると、CLOS（Bobrow et al. [47]; Kiczales et al. [262]）やCecil（Chambers [94]; Chambers and Leavens [96]）などの言語で使われている**多重メソッドディスパッチ**の基礎となる。この機構はCastagna, Ghelli, and Longo [93]（Castagna [92]も参照）によるλ-&計算によって形式化されている。
- アドホック多相をより強力にした**内包的多相**（Harper and Morrisett [198]; Crary, Weirich, and Morrisett [124]）では、実行時の型に関する制限された計算が可能になる。内包的多相は、多相的な言語のさまざまな発展的実装技法を可能にする技術である。例えば、タグなしガベージコレクション、「アンボックス化」された関数引数、多相マーシャリング、「平坦化」された空間的に効率の良いデータ構造などが可能になる。
- アドホック多相をさらに強力にした形には、`typecase` プリミティブからなるものもある。これは実行時の型情報に基づいた任意のパターンマッチを可能にする（Abadi, Cardelli, Pierce, and Rémy [8]; Abadi, Cardelli, Pierce, and Plotkin [7]; Henglein [203]; Leroy and Mauny [281]; Thatte [459]）。Java の `instanceof` 検査のような言語機能は、`typecase` を制限したものと考えられる。
- 第 15 章の**部分型多相**は、包摂規則を使って一つの項に複数の型を与える。これにより項の振る舞いに関する情報を選択的に「忘れる」ことが可能になる。

以上の分類は排他的ではない。一つの言語に複数の多相性を盛り込むことができる。例えばStandard ML は、パラメータ多相と、組み込み算術演算の単純なオーバーロードとの両方を提供しているが、部分型付けは行わない。一方 Java は、部分型付け、オーバーロード、単純なアドホック多相（`instanceof`）を持っているが、（これを執筆している時点では）パラメータ多相を持っていない。Java にパラメータ多相を追加する提案は幾つかあり、最も有名なのが GJ（Bracha, Odersky, Stoutamire, and Wadler [54]）である[†1]。

単に「多相性」というと、異なるプログラミング言語のコミュニティ間でかなりの誤解が生じる。関数的プログラミングの分野（つまり ML や Haskell のような言語を使ったり設計したりしているコミュニティ）では、多相性というとほぼ常にパラメータ多相を意味する。しかしオブジェクト指向プログラミングの分野ではほぼ常に部分型多相を意味し、パラメータ多相のことは**総称性**（または**ジェネリクス**）と呼ぶ。

23.3 System F

本章で学ぶ体系は、一般に **System F** と呼ばれる。これは最初に論理学の証明論を背景として、Jean-Yves Girard [177] が発見した。程なくして、計算機科学者 John Reynolds [408] が本質的に同等な能力を持つ型システムを独立に提案した。Reynolds はこれを**多相ラムダ計算**と呼んだ。この体系は多相型の基礎研究の題材として、また、数多くのプログラミング言語設計の土台として広く使われている。System F はしばしば**二階ラムダ計算**とも呼ばれる。**Curry-Howard 対応**を通じて、二階直観論理に対応するためである。二階直観論理とは、個体（項）だけでなく述語（型）の量

[†1] （訳注）その後、Featherweight Java（第 19 章参照）に基づく形式化を経て、総称型として Java 5.0 に導入された。

化も許している論理体系である。

System F は、$\lambda_{\rightarrow}$（単純型付きラムダ計算）を単純に拡張して定義される。$\lambda_{\rightarrow}$ には、項の中から項を抽象化するためのラムダ抽象と、抽象化された部分に値を与える適用があった。ここで我々は項の中から**型**を抽象化して後で埋める機構が欲しいわけなので、λX.t という抽象の新しい形式（ただしパラメータ X は型）と、t [T] という適用の新しい形式（ただし引数 T は型式）を導入する。この新しい抽象を**型抽象**と呼び、新しい適用を**型適用**または**具体化**と呼ぶ。

$\lambda_{\rightarrow}$ と同様、評価中に型抽象が型適用と合わさると、簡約基をなす。簡約規則

$$(\lambda X.t_{12})\ [T_2] \longrightarrow [X \mapsto T_2]t_{12} \qquad \text{(E-TAPPTABS)}$$

を新たに導入する。これは普通の抽象と適用のための簡約規則

$$(\lambda x{:}T_{11}.t_{12})\ v_2 \longrightarrow [x \mapsto v_2]t_{12} \qquad \text{(E-APPABS)}$$

と類似した規則である。例えば、**多相恒等関数**

```
id = λX. λx:X. x;
```

は、id [Nat] と書くことで Nat に適用できる。その結果は [X ↦ Nat](λx:X.x) で、つまり λx:Nat.x である。これは自然数における恒等関数である。

最後に、多相的な抽象の**型**を決める必要がある。λx:Nat.x のような普通の関数を分類するには、Nat→Nat のような型を使った。今必要なのは、id のように多相的な関数を分類するための、定義域が型であるような別種の「矢印型」である。id は任意の引数 T に適用されたとき、型 T→T の関数となることに注目しよう。つまり、id の結果の型は、id に引数として与える実際の型に依存するということである。この依存性を表すため、id の型を ∀X.X→X と書く。多相的な抽象と適用の型付け規則は、項レベルの抽象と適用の規則に似ている。

$$\frac{\Gamma, X \vdash t_2 : T_2}{\Gamma \vdash \lambda X.t_2 : \forall X.T_2} \qquad \text{(T-TABS)}$$

$$\frac{\Gamma \vdash t_1 : \forall X.T_{12}}{\Gamma \vdash t_1\ [T_2] : [X \mapsto T_2]T_{12}} \qquad \text{(T-TAPP)}$$

t_2 の部分導出で使われる文脈に型変数 X が含まれていることに注意されたい。慣習 5.3.4 は引き続き適用される。つまり、（項や型の）変数名は、既に Γ に束縛されているどの名前とも異なるように選ばれ[†2]、またラムダによって束縛された型変数名も、この条件を満たすよう自由に付け替え可能である。（このように文脈の構成方法に規則を導入する以外に、フレッシュさの条件を規則 T-TABS の明示的な付帯条件として与えているような System F の方式もある。）今のところ文脈に型変数を入れる役割は、スコープの記録を付け、同じ型変数を文脈に二回追加してしまわないようにすることだけである。後の章では、**上界**（第 26 章）や**カインド**（第 29 章）など、型変数にさまざまな情報を付加する。

図 23-1 に多相ラムダ計算の完全な定義を示す。$\lambda_{\rightarrow}$ との違いには網掛けを施してある。例によって、これは純粋な計算体系だけを定義しており、レコードなど他の型構築子、Nat や Bool などの基本型、let や fix など項の言語に対する拡張などは省略されている。これらの構文は、純粋な体系に単純に追加できるので、以後の例でも遠慮なく用いる。

[†2] （訳注：原著正誤表より）Γ, x:T という形の文脈は、T に現れる自由な型変数がすべて Γ で束縛されているときに限り正しい形式である。

$\rightarrow \forall$ 　　　　　　　　$\lambda_{\rightarrow}$（図 9-1）に基づく

構文

t ::=		項：
	x	変数
	$\lambda x{:}T.t$	ラムダ抽象
	$t\ t$	関数適用
	$\lambda X.t$	型抽象
	$t\ [T]$	型適用
v ::=		値：
	$\lambda x{:}T.t$	ラムダ抽象値
	$\lambda X.t$	型抽象値
T ::=		型：
	X	型変数
	$T \rightarrow T$	関数の型
	$\forall X.T$	全称型
Γ ::=		文脈：
	$\varnothing$	空の文脈
	$\Gamma, x{:}T$	項変数の束縛
	Γ, X	型変数の束縛

評価 　$t \longrightarrow t'$

$$\frac{t_1 \longrightarrow t_1'}{t_1\ t_2 \longrightarrow t_1'\ t_2} \quad \text{(E-App1)}$$

$$\frac{t_2 \longrightarrow t_2'}{v_1\ t_2 \longrightarrow v_1\ t_2'} \quad \text{(E-App2)}$$

$$(\lambda x{:}T_{11}.t_{12})\ v_2 \longrightarrow [x \mapsto v_2]t_{12} \quad \text{(E-AppAbs)}$$

$$\frac{t_1 \longrightarrow t_1'}{t_1\ [T_2] \longrightarrow t_1'\ [T_2]} \quad \text{(E-TApp)}$$

$$(\lambda X.t_{12})\ [T_2] \longrightarrow [X \mapsto T_2]t_{12} \quad \text{(E-TappTabs)}$$

型付け 　$\Gamma \vdash t : T$

$$\frac{x{:}T \in \Gamma}{\Gamma \vdash x : T} \quad \text{(T-Var)}$$

$$\frac{\Gamma, x{:}T_1 \vdash t_2 : T_2}{\Gamma \vdash \lambda x{:}T_1.t_2 : T_1 \rightarrow T_2} \quad \text{(T-Abs)}$$

$$\frac{\Gamma \vdash t_1 : T_{11} \rightarrow T_{12} \qquad \Gamma \vdash t_2 : T_{11}}{\Gamma \vdash t_1\ t_2 : T_{12}} \quad \text{(T-App)}$$

$$\frac{\Gamma, X \vdash t_2 : T_2}{\Gamma \vdash \lambda X.t_2 : \forall X.T_2} \quad \text{(T-TAbs)}$$

$$\frac{\Gamma \vdash t_1 : \forall X.T_{12}}{\Gamma \vdash t_1\ [T_2] : [X \mapsto T_2]T_{12}} \quad \text{(T-TApp)}$$

▲ 図 23-1.　多相ラムダ計算（System F）

23.4　事例

本節では、多相性を使ったプログラミングの事例を幾つか考えていく。肩慣らしとして幾つかの、小さいながらも段々ややこしくなる例を見てみよう。これで System F の表現力がいくらかわかるだろう。それから、リストや木などの「通常」の多相的なプログラミングの基本的なアイディアについて考える。最後の二つの小節では、第 5 章の型無しラムダ計算で説明した、ブール値、数、リストのような単純な代数的データ構造の Church 表現の、型付き版を紹介する。Church 表現は実用的見地からはそれほど重要でないが（高水準なプログラミング言語には、ブール値や数といった重要な機能はプリミティブとして組み込んだほうがコンパイルで良いコードを生成しやすくなる）、System F の複雑さと強力さを理解するには絶好の演習となる。第 24 章では、**モジュラープログラミング**や**抽象データ型**といった領域のために多相性を応用する事例を考える。

肩慣らし

型抽象と適用を使って単一の多相恒等関数を定義する方法は既に述べた。

```
  id = λX. λx:X. x;
▶ id : ∀X. X → X
```

またそれから、必要に応じて各型のための具体的な恒等関数に具体化する方法も述べた。

```
  id [Nat];
▶ <fun> : Nat → Nat

  id [Nat] 0;
▶ 0 : Nat
```

より便利な例として多相的な `double` 関数を挙げる。

```
  double = λX. λf:X→X. λa:X. f (f a);
▶ double : ∀X. (X→X) → X → X
```

型 X を抽象化しているので、異なる型引数を与えて `double` を具体化することで、個別の型に特化した `double` 関数を得られる。

```
  doubleNat = double [Nat];
▶ doubleNat : (Nat→Nat) → Nat → Nat

  doubleNatArrowNat = double [Nat→Nat];
▶ doubleNatArrowNat : ((Nat→Nat)→Nat→Nat) →
                       (Nat→Nat) → Nat → Nat
```

いったん `double` を型引数で具体化したら、相応の型を持つ具体的な関数・引数に対して適用できるようになる。

```
  double [Nat] (λx:Nat. succ(succ(x))) 3;
▶ 7 : Nat
```

少しややこしい例として、多相的な自己適用を挙げる。単純型付きラムダ計算では、`λx. x x` という型無し項を型付けできなかったことを思い出そう（演習 9.3.2）。しかし System F では、`x` に多相型を与え、適切に具体化することで、この項が型付け可能になる。

```
  selfApp = λx:∀X.X→X.  x [∀X.X→X] x;
▶ selfApp : (∀X. X→X) → (∀X. X → X)
```

自己適用の、（多少は）便利といえる例を示す。多相的な `double` 関数をそれ自身に適用すると、多相的な四倍関数が得られる。

```
  quadruple = λX. double [X→X] (double [X]);
▶ quadruple : ∀X. (X→X) → X → X
```

演習 23.4.1. [★ ↛]　図 23-1 の型付け規則を使って、上記の項が与えられた型を持つことを確認せよ。 □

多相リスト

多相性を使った現実世界のプログラミングは、以上のようなややこしい例よりずっと平凡である。単純な多相的プログラミングの例として、現状のプログラミング言語に、型構築子 `List` と通常の

リスト操作プリミティブの項構築子を、以下のような型として持たせることを考えよう。

```
▸ nil : ∀X. List X
  cons : ∀X. X → List X → List X
  isnil : ∀X. List X → Bool
  head : ∀X. List X → X
  tail : ∀X. List X → List X
```

最初にリストを導入したとき（11.12 節）は、任意の型の要素を持つリストに対して操作を適用できるよう、推論規則を「特注」した。今回は操作に、全く同じ制約を表現する多相型を与えられる。つまり、もはやリストを核言語に「焼き付ける」必要はなく、単に特定の多相型を持つ定数を数個提供するライブラリとみなせる。第 13 章の参照セルで使った Ref 型やプリミティブ操作など、他の多くの一般的なデータ構造や制御構造に対しても同様である。

以上のプリミティブは、リストへの多相的な操作を独自に定義するために使える。例えば以下は多相的な map 関数である。すなわち、X から Y への関数と型 X の要素を持つリストを受け取り、型 Y の要素を持つリストを返す。

```
  map = λX. λY.
          λf: X→Y.
            (fix (λm: List X → List Y.
                    λl: List X.
                      if isnil [X] l
                        then nil [Y]
                        else cons [Y] (f (head [X] l))
                                      (m (tail [X] l))));
▸ map : ∀X. ∀Y. (X→Y) → List X → List Y

  l = cons [Nat] 4 (cons [Nat] 3 (cons [Nat] 2 (nil [Nat])));
▸ l : List Nat

  head [Nat] (map [Nat] [Nat] (λx:Nat. succ x) l);
▸ 5 : Nat
```

演習 23.4.2. [★ ↛]　map が本当に上記の型を持つことを確認せよ。 □

演習 23.4.3. [推奨, ★★]　map を参考に、多相的なリスト反転関数を書け。

```
reverse : ∀X. List X → List X
```

この演習では実際に手を動かしてみるのが一番だ。fullomega 検査器を使い、fullomega ディレクトリのファイル test.f の内容を入力ファイル先頭にコピーして使うとよい。（そこには List 構築子とそれに関係する操作の定義があるが、その定義は第 29 章で議論する System F_ω の強力な抽象化機能を必要とする。この演習では、これらがどのように働くか正確に理解する必要はない。）

□

演習 23.4.4. [★★ ↛]　単純な多相ソート関数を書け。

```
sort : ∀X. (X→X→Bool) → List X → List X
```

第一引数は型 X の要素の比較関数である。 □

Church 表現

5.2 節では、ブール値、自然数、リストなどのさまざまなプリミティブデータ値が純粋な型無しラムダ計算の関数として表現できることを述べた。本節では、これら **Church 表現**が System F でも同様に実現できることを示す。必要なら 5.2 節を見て Church 表現に関する直観を新たにしておくとよい。

Church 表現は二つの理由で興味深い。まず、型抽象と適用についての理解を深める練習になる。そして純粋な型無しラムダ計算と同様に、純粋な体系で多くのデータ構造や制御構造が表現できるという意味で、System F が計算的にとても強力であることを実感できる。つまり System F を核として本格的なプログラミング言語を設計する際、核言語の本質を変えることなく、(効率化や、その結果としてより便利な具象構文を持たせるために) これらの機能をプリミティブとして追加できるということである。もちろん、興味深い高水準な言語機能すべてがそのようにできるわけではない。例えば、第 13 章で $\lambda_{\to}$ に**参照**を追加したように、System F に参照を追加すると、型システムの計算的性質が本質的に変化してしまう。

Church ブール値から始めよう。型無しラムダ計算では、ブール値の定数 `true` と `false` を、以下のラムダ項 `tru` と `fls` で表現したことを思い出そう。

```
tru = λt. λf. t;
fls = λt. λf. f;
```

これらの項はどちらも二つの引数を取り、そのいずれかを返す。もし `tru` と `fls` に共通の型をあてたいなら、その引数が同じ型を持つと仮定するとよい (呼び出し元は、自分が扱っているのが `tru` なのか `fls` なのか知らないだろうから)。しかし、その型自体は任意でよい (`tru` と `fls` は、引数のいずれかを返すということ以外、引数に対して何もする必要がないから)。以上より、`tru` と `fls` は以下の型になる。

```
CBool = ∀X.X→X→X;
```

System F の項 `tru` と `fls` は、上記の型無し版に適切に型抽象を与えることで得られる。

```
  tru = λX. λt:X. λf:X. t;
▸ tru : CBool

  fls = λX. λt:X. λf:X. f;
▸ fls : CBool
```

`not` のような一般的なブール演算は、元のブール値を使ってどちらの引数を返すか決めるような、新しいブール値を作ることで定義できる。

```
  not = λb:CBool. λX. λt:X. λf:X. b [X] f t;
▸ not : CBool → CBool
```

演習 23.4.5. [推奨, ★] 型 `CBool` の引数を二つ取り、その論理積を計算する項 `and` を書け。 □

数も同様に行ける。5.2 節で導入された **Church 数**は、各自然数 `n` を表現するものとして、二つの引数 `s` と `z` を取り `z` に `s` を `n` 回適用する関数を使った。

```
c0 = λs. λz. z;
c1 = λs. λz. s z;
c2 = λs. λz. s (s z);
c3 = λs. λz. s (s (s z));
```

引数 `z` は `s` の定義域と同じ型を持つべきで、`s` の返り値も同じ型を持つべきことは明らかである。

したがって、System F の Church 数の型は以下のようになる。

```
CNat = ∀X. (X→X) → X → X;
```

この型の要素は、型無し Church 数に適切に注釈をすると得られる。

```
  c0  =  λX. λs:X→X. λz:X. z;
▶ c0 : CNat

  c1  =  λX. λs:X→X. λz:X. s z;
▶ c1 : CNat

  c2  =  λX. λs:X→X. λz:X. s (s z);
▶ c2 : CNat
```

Church 数における後者関数の型付き版は以下のように定義できる。

```
  csucc = λn:CNat. λX. λs:X→X. λz:X. s (n [X] s z);
▶ csucc : CNat → CNat
```

つまり、csucc n は CNat の要素を返す。その返り値は、s と z が与えられると、(n を s と z に適用することで) s を z に n 回適用し、さらにもう一回適用するような項である。他の算術演算も同様に定義できる。例えば加算は後者関数を使って

```
  cplus = λm:CNat. λn:CNat. m [CNat] csucc n;
▶ cplus : CNat → CNat → CNat
```

と定義できる。より直接的に、

```
  cplus = λm:CNat. λn:CNat.  λX. λs:X→X. λz:X. m [X] s (n [X] s z);
▶ cplus : CNat → CNat → CNat
```

でもよい。もし言語にプリミティブな数 (図 8-2) があるなら、以下の関数を使って、Church 数から通常のプリミティブな数へ変換できる。

```
  cnat2nat = λm:CNat. m [Nat] (λx:Nat. succ(x)) 0;
▶ cnat2nat : CNat → Nat
```

これを使って、Church 数上で定義した演算が望み通りの算術関数を計算していることを確かめられる。

```
  cnat2nat (cplus (csucc c0) (csucc (csucc c0)));
▶ 3 : Nat
```

演習 23.4.6. [推奨, ★★]　Church 数 c_0 に適用されたときは tru を返し、さもなくば fls を返す関数 iszro を定義せよ。 □

演習 23.4.7. [★★ ↛]　以下の項

```
  ctimes = λm:CNat. λn:CNat.  λX. λs:X→X. n [X] (m [X] s);
▶ ctimes : CNat → CNat → CNat

  cexp = λm:CNat. λn:CNat.  λX. n [X→X] (m [X]);
▶ cexp : CNat → CNat → CNat
```

が、示された通りの型を持つことを確かめよ。これらが算術的な乗算と冪乗演算になっていることについて非形式的な論証を与えよ。 □

演習 23.4.8. [推奨, ★★] 以下の型

```
PairNat = ∀X. (CNat→CNat→X) → X;
```

が数の二つ組を表現していることを、以下の三つの関数を書くことで示せ。

```
pairNat : CNat→CNat→PairNat;
fstNat : PairNat→CNat;
sndNat : PairNat→CNat;
```

最初の関数は二つの数からこの型の要素を作る。他の二つはそれぞれ第一、第二要素を読み出す。
□

演習 23.4.9. [推奨, ★★★] 演習 23.4.8 で定義した関数を使い、Church 数の前者値を計算する関数 `prd` を書け（`0` が入力されたら `0` を返せ）。ヒント：鍵となるアイディアは 46 ページの例で述べている。二つ組 (i, j) から $(i+1, i)$ へ写す関数 `f : PairNat→PairNat` を定義せよ。つまり、引数の第二要素を捨てて、第一要素を第二要素にコピーし、第一要素に 1 を足して返すものである。二つ組 $(0, 0)$ から始め、これに `f` を `n` 回適用すると、二つ組 $(n, n-1)$ が得られる。この第二要素を射影することで n の前者値を取り出せる。 □

演習 23.4.10. [★★★] Church 数の前者関数を計算する方法は他にもある。`k` が型無しラムダ項 `λx. λy. x` を表し、`i` が `λx. x` を表すとする。以下の型無しラムダ項

```
vpred = λn. λs. λz. n (λp. λq. q (p s)) (k z) i
```

(Barendregt [40] から引用。Barendregt はこれを J. Velmans によるものとしている）は、型無し Church 数の前者値を計算する。この項に型抽象や適用を必要に応じて追加し、さらに型無し項の束縛変数に適切な型を注釈することで、この項が System F で型付けできることを示せ。追加の課題として、なぜ動くのかを説明してみよ！ □

リストの表現

最後の例として、数の Church 表現をリストに拡張してみよう。これは、純粋な System F の表現力がわかる好例である。多相的なリストの操作に関するプログラミング例を前述したが、そのすべてが純粋な言語だけで実際に表現できるからである。（簡便さのため、一般的な再帰関数を定義するために `fix` 構文を使うが、`fix` なしでも本質的に同じ構成が実現可能である。演習 23.4.11 や演習 23.4.12 を参照されたい。）

演習 5.2.8 では、型無しラムダ計算において、自然数の表現と極めてよく似た方法でリストが表現できることを説明した。一進表記された数は、実質的にダミーの要素を持つリストのようなものである。この考え方を任意の型の要素に一般化するとリストの Church 表現になる。要素 `x`、`y`、`z` を持つリストは、任意の関数 `f` と初期値 `v` が与えられ、`f x (f y (f z v))` を計算する関数として表現される。OCaml の用語でいうと、リストは自分自身の `fold_right` 関数として表現される。

型 `X` の要素を持つリストの型 `List X` は以下のように定義される。

```
List X = ∀R. (X→R→R) → R → R;
```

このリスト表現での `nil` 値は簡単に書ける[†3]。

[†3] ここに `as` の注釈を入れると、型検査器は `nil` の型を読みやすい形で出力できるようになる。11.4 節で述べたように、本書で用いるすべての型検査器は型を表示する前に単純な略記化を行うが、この略記関数は `List` のような「パラメータ付き省略」を自動的に扱えるほどは賢くない。

```
  nil = λX. (λR. λc:X→R→R. λn:R. n) as List X;
▸ nil : ∀X. List X
```

cons と isnil の操作も簡単である。

```
  cons = λX. λhd:X. λtl:List X.
           (λR. λc:X→R→R. λn:R. c hd (tl [R] c n)) as List X;
▸ cons : ∀X. X → List X → List X

  isnil = λX. λl:List X. l [Bool] (λhd:X. λtl:Bool. false) true;
▸ isnil : ∀X. List X → Bool
```

head の操作は少し難しい。まず、空のリストの head をどう扱うかが難しい。これを解決するには、言語に一般的な不動点演算子があればそれを使って任意の型の式を作れることを思い出すとよい。実際さらに、型抽象を使えば、型 X を受け取り、Unit から X への関数で unit に適用されたら発散する関数を返すような、単一の一様な関数を書ける。

```
  diverge = λX. λ_:Unit. fix (λx:X. x);
▸ diverge : ∀X. Unit → X
```

これで、head [X] nil の「結果」として diverge [X] unit を使える。

```
  head = λX. λl:List X. l [X] (λhd:X. λtl:X. hd) (diverge [X] unit);
▸ head : ∀X. List X → X
```

残念ながら、この定義は望むものにはまだなっていない。たとえ空でないリストに適用されたときでも**常に**発散してしまう。正しい振る舞いにするには、diverge [X] が l の引数として与えられる時点では、Unit 型の引数が実際には渡らないように、少し書き直す必要がある。これには、引数 unit を消して、それに合わせ l への第一引数の型を変えればよい。

```
  head =
    λX. λl:List X.
      (l [Unit→X] (λhd:X. λtl:Unit→X. λ_:Unit. hd) (diverge [X]))
      unit;
▸ head : ∀X. List X → X
```

つまり l は、型 X→(Unit→X)→(Unit→X) の関数と型 Unit→X の基底値に適用され、型 Unit→X の関数を作る。l が空のリストである場合は、この結果は diverge [X] となる。l が空でないリストである場合は、この結果は unit を受け取って l の先頭の要素を返す関数となる。実際の先頭要素を得られるよう（あるいは、運が悪ければ発散するよう）、最後に l の結果が unit に適用される。これで head が期待通りの型を持つようになる。

tail 関数のために Pair X Y という略記を用いる。これは演習 23.4.8 の PairNat 型を一般化したもので、第一要素が型 X、第二要素が型 Y の二つ組の Church 表現の型を表す。

```
  Pair X Y = ∀R. (X→Y→R) → R;
```

この二つ組への操作は、上記の型 PairNat の操作を単純に一般化したものである。

```
▸ pair : ∀X. ∀Y. X → Y → Pair X Y
  fst : ∀X. ∀Y. Pair X Y → X
  snd : ∀X. ∀Y. Pair X Y → Y
```

これで、tail 関数が以下のように書ける。

```
tail =
  λX. λl: List X.
    (fst [List X] [List X] (
       l [Pair (List X) (List X)]
         (λhd: X. λtl: Pair (List X) (List X).
           pair [List X] [List X]
             (snd [List X] [List X] tl)
             (cons [X] hd (snd [List X] [List X] tl)))
         (pair [List X] [List X] (nil [X]) (nil [X]))));
▸ tail : ∀X. List X → List X
```

演習 23.4.11. [★★] 厳密に言うと、この小節の例は純粋な System F で表現されていない。空のリストに head が適用されたときの「返り値」を作るのに `fix` 演算子を使っているからである。head の別のバージョンとして、(発散する代わりに) リストが空だったときに返すべき値を引数として受け取るものを書け。 □

演習 23.4.12. [推奨, ★★★] (`fix` がない) 純粋な System F で、以下の型を持つ関数 `insert` を書け。

$$\forall X.\ (X{\to}X{\to}\mathtt{Bool}) \to \mathtt{List}\ X \to X \to \mathtt{List}\ X$$

これは、比較関数、ソートされたリスト、新しい要素の三つを受け取り、新しい要素をリストの適切な位置に (つまりその要素より小さな要素より後に) 挿入する関数である。次にこの `insert` を使って、純粋な System F でリストのソート関数を作れ。 □

23.5 基本的な性質

System F の基本的な性質は単純型付きラムダ計算の性質によく似ている。特に型保存と進行の証明は第 9 章で示したものの単純な拡張である。

定理 23.5.1. [**保存**] $\Gamma \vdash \mathtt{t} : \mathtt{T}$ かつ $\mathtt{t} \longrightarrow \mathtt{t}'$ ならば $\Gamma \vdash \mathtt{t}' : \mathtt{T}$ □

[証明] 演習[推奨, ★★★]。 □

定理 23.5.2. [**進行**] t が閉じた、正しく型付けされた項ならば、t は値であるか、さもなくばある t′ が存在して $\mathtt{t} \longrightarrow \mathtt{t}'$ となる。 □

[証明] 演習[推奨, ★★★]。 □

System F には、$\lambda_{\to}$ と同様に**正規化**の性質、つまりすべての正しく型付けされたプログラムの評価は停止するという事実がある[†4]。上記の型安全性の定理とは異なり、正規化の証明は大変難しい。(実際、演習 23.4.12 で行ったように、`fix` に頼ることなく純粋な言語でソート関数のようなものが書けることを考えると、正規化が成り立つことはなかなか驚きに値する。) この証明は第 12 章で説明した方法の一般化に基づくもので、Girard の博士論文 [177] (Girard, Lafont, and Taylor [179] も参照) の主要成果である。それ以来、Girard の証明技法は多くの人に解析・改変されてきた (Gallier [165] を見よ)。

[†4] 実際、より制限の少ない、完全ベータ簡約に基づいた操作的意味論の下でも、System F は**強正規化**可能性を持つ。すなわち、正しく型付けされた項からのすべての簡約パスは停止することが保証される。

定理 23.5.3.［正規化］　正しく型付けされた System F の項は正規化可能である。□

23.6 型消去、型付け可能性、型再構築

9.5 節で $\lambda_{\rightarrow}$ に対して行ったように、System F の項から（すべての型抽象・適用を含む）型注釈をすべて取り払って型無しラムダ項にする**型消去**（type erasure）関数を定義できる。

$$\begin{array}{lcl} \mathit{erase}(\mathtt{x}) & = & \mathtt{x} \\ \mathit{erase}(\lambda\mathtt{x}\mathtt{:}\mathtt{T}_1\mathtt{.}\ \mathtt{t}_2) & = & \lambda\mathtt{x}\mathtt{.}\ \mathit{erase}(\mathtt{t}_2) \\ \mathit{erase}(\mathtt{t}_1\ \mathtt{t}_2) & = & \mathit{erase}(\mathtt{t}_1)\ \mathit{erase}(\mathtt{t}_2) \\ \mathit{erase}(\lambda\mathtt{X}\mathtt{.}\ \mathtt{t}_2) & = & \mathit{erase}(\mathtt{t}_2) \\ \mathit{erase}(\mathtt{t}_1\ [\mathtt{T}_2]) & = & \mathit{erase}(\mathtt{t}_1) \end{array}$$

正しく型付けされた項 $\mathtt{t}$ が存在して $\mathit{erase}(\mathtt{t}) = \mathtt{m}$ であるとき、型無しラムダ計算の項 $\mathtt{m}$ は System F で**型付け可能**であるという。**型再構築**問題は、型無し項 $\mathtt{m}$ が与えられたとき、型消去すると $\mathtt{m}$ になるような正しく型付けされた項を見つけられるかどうか、という問題である。

System F の型再構築は、プログラミング言語の論文で最も長く扱われ続けた問題の一つであった。1970 年代前半から未解決だった問題で、1990 年代前半になり、Wells によってついに（否定的に）解決された。

定理 23.6.1.［Wells, 1994 [492]］　型無しラムダ計算の閉じた項 $\mathtt{m}$ が与えられたとき、System F の正しく型付けされた項 $\mathtt{t}$ で $\mathit{erase}(\mathtt{t}) = \mathtt{m}$ となるようなものが存在するかどうかは決定不能である。□

System F では完全な型再構築だけでなく、部分的な型再構築もさまざまなものが決定不能だとわかっている。例えば、以下のような「部分的消去」関係を考えよう。これは型適用の引数を（部分的に）消去し、それ以外の型注釈をすべて残す。部分的に消去された項の中でも、型適用が消去された**位置**は空の角括弧で印付けられていることに注意されたい。

$$\mathtt{x} > \mathtt{x} \qquad \frac{\mathtt{t}_2 > \mathtt{t}_2'}{\lambda\mathtt{x}\mathtt{:}\mathtt{T}_1\mathtt{.}\ \mathtt{t}_2 > \lambda\mathtt{x}\mathtt{:}\mathtt{T}_1\mathtt{.}\ \mathtt{t}_2'} \qquad \frac{\mathtt{t}_2 > \mathtt{t}_2'}{\lambda\mathtt{x}\mathtt{:}\mathtt{T}_1\mathtt{.}\ \mathtt{t}_2 > \lambda\mathtt{x}\mathtt{.}\ \mathtt{t}_2'} \qquad \frac{\mathtt{t}_1 > \mathtt{t}_1' \qquad \mathtt{t}_2 > \mathtt{t}_2'}{\mathtt{t}_1\ \mathtt{t}_2 > \mathtt{t}_1'\ \mathtt{t}_2'}$$

$$\frac{\mathtt{t}_2 > \mathtt{t}_2'}{\lambda\mathtt{X}\mathtt{.}\ \mathtt{t}_2 > \lambda\mathtt{X}\mathtt{.}\ \mathtt{t}_2'} \qquad \frac{\mathtt{t}_1 > \mathtt{t}_1'}{\mathtt{t}_1\ [\mathtt{T}_2] > \mathtt{t}_1'\ [\mathtt{T}_2]} \qquad \frac{\mathtt{t}_1 > \mathtt{t}_1'}{\mathtt{t}_1\ [\mathtt{T}_2] > \mathtt{t}_1'\ [\,]}$$

定理 23.6.2.［Pfenning, 1993 [357]］　$\mathtt{s}$ が与えられたとき、System F の正しく型付けされた項 $\mathtt{t}$ で、$\mathtt{t} > \mathtt{s}$ となるものが存在するかどうかは決定不能である。□

Boehm [48; 49] の初期の観察に基づいて、Pfenning はこの形の型再構築が高階単一化（既に決定不能であることが知られていた）と同等に難しいことを示した。この否定的な結果がそのまま、高階単一化の効率的な半アルゴリズムに関する Huet の既存の業績 [227] に基づいて、部分的な型再構築の使いものになる技法（Pfenning [355; 357]）につながったことは興味深い。この流れに続く後の改良には、より洗練された高階制約解消アルゴリズム（Dowek, Hardin, Kirchner, and Pfenning [139]）がある。これは非停止性や一意でない解の生成といった、面倒な可能性を排除している。LEAP（Pfenning and Lee [363]）、Elf（Pfenning [356]）、FX（O'Toole and Gifford [348]）などの言語で関連アルゴリズムが使われた経験から、実用上はかなりよく振る舞うことがわかっている。

一級存在型（第 24 章を見よ）が ML の `datatype` 機構と統合できるという Perry の観察がきっかけとなって、部分的な型再構築への異なるアプローチが見つかった。このアイディアは Läufer

と Odersky（Läufer [272]; Läufer and Odersky [273]）によって詳しく研究された。本質的に、datatype 構築子と分解子は、値が直和型に入射される、あるいは直和型から射影されるはずの箇所、再帰型が fold/unfold されるはずの箇所、（存在型がある場合は）パッケージ化/アンパッケージ化が行われるはずの箇所に付けられる、明示的な型注釈であるとみなせる。このアイディアは Rémy [404] によって一級（非可述的）全称量化子を持つように拡張された。より最近では、Odersky and Läufer [342] が提案し Garrigue and Rémy [170] が発展させた研究がある。プログラマが関数の引数の型を明示的に注釈できるようにして、ML 式の型再構築を保守的に拡張したものである。これは（自動的に推論できる注釈と異なり）埋め込まれた全称量化子を持つことが許され、これによって ML と、より強力な非可述的体系との乖離を一部橋渡しする。この系統の型再構築アプローチは、比較的単純で、ML の多相性と明快に結合できるという利点がある。

Pierce と Turner は、**部分型付け**と非可述的多相の両方を持つ体系における、部分的な型再構築に対する実用的なアプローチ（**局所型推論**や**局所型再構築**といわれる）を提案した [375]（Pierce and Turner [374]; Hosoya and Pierce [220] も見よ）。局所的型推論は、GJ（Bracha, Odersky, Stoutamire, and Wadler [54]）や Funnel（Odersky and Zenger [345]）など、最近の言語設計に幾つか見られる。後者は**色付き局所型推論**（Odersky, Zenger, and Zenger [346]）といわれる、より強力な方法を導入している。

より単純だが、挙動を予想するのが難しい**貪欲型推論**アルゴリズムは Cardelli [81] が提案した。似たアルゴリズムが NuPRL （Howe [225]）や Lego （Pollack [389]）といった、依存型理論の証明検査器で使われている。このアイディアは、プログラマが任意の型注釈を省略しうるというものである。それぞれの省略箇所では、構文解析器によってフレッシュな単一化変数 X が生成される。型検査の間、部分型検査アルゴリズムが必要に応じて、ある型 S が T の部分型であるかどうかを調べる。ただし、S と T は共に単一化変数を含みうる。部分型の検査は、部分ゴールが X <: T または T <: X になるまでは従来通りに進む。この形になったら、この制約を満たす最も単純な方法として、X を T に具体化する。しかし X を T にすることが、取りうる中で最適な選択とは限らず、この選択のせいで後々 X を含む型の部分型検査が失敗してしまうかもしれない（別の選択なら成功していたかもしれない場合でも）。ただし、Pfenning 他の不完全な型再構築と同様、Cardelli の実装や Pict 言語（Pierce and Turner [376]）の初期版でこのアルゴリズムが使われた経験からいうと、このアルゴリズムの貪欲な選択は、ほとんどの場合においては正しい。だがうまくいかなかった場合、貪欲アルゴリズムの挙動のせいで、最適ではない具体化がなされた箇所から遠く離れたところで摩訶不思議なエラーが発生し、プログラマがひどく困惑することがある。

演習 23.6.3. [★★★★] 正規化可能性は、型無し項 omega = (λx. x x) (λy. y y) が System F で型付けできないことを意味している。omega の簡約は決して正規形に至らないからである。しかしこの事実は、型付け関係を定義する規則だけを使って、もっと直接、「組み合わせ」的に証明できる。

(1) System F の項が、変数、または抽象 λx:T.t、または適用 t s であるとき（つまり、型抽象 λX.t でも型適用 t [S] でも**ない**場合）、その項は**露出している**という。
t が（何らかの文脈で）正しく型付けされ、かつ *erase*(t) = m であるならば、ある露出している項 s があり、*erase*(s) = m かつ s は（拡張されているかもしれない文脈で）正しく型付けされる、ということを示せ。

(2) $\lambda X_1 \ldots \lambda X_n$.t という形で型抽象が入れ子になった列を、$\lambda\overline{X}$.t と略記する。同様に ((t [$A_1$]) ... [$A_{n-1}$]) [$A_n$] という型適用が入れ子になった列を t [$\overline{A}$] と書く。さらに $\forall X_1 \ldots \forall X_n$.T という多相型が入れ子になった列を $\forall\overline{X}$.T と書く。これらの列は空でもよいことに注意されたい。例えば $\overline{X}$ が型変数の空の列だったら、$\forall\overline{X}$.T はただの T である。

erase(t) = m かつ $\Gamma \vdash$ t : T ならば、$\lambda\overline{X}$. (u $[\overline{A}]$) という形であるような s が存在し、*erase*(s) = m かつ $\Gamma \vdash$ s : T となることを示せ。ただし $\overline{X}$ は型変数の列、$\overline{A}$ は型の列、u は露出している項であるとする。

(3) t が（Γ の下で）型 T であるような露出した項であり、*erase*(t) = m n のとき、t は s u という形であり、$\Gamma \vdash$ s : U→T かつ $\Gamma \vdash$ u : U であることを示せ。ただし s と u は *erase*(s) = m かつ *erase*(u) = n であるような項である。

(4) x:T $\in \Gamma$ とする。$\Gamma \vdash$ u : U かつ *erase*(u) = x x ならば、以下のいずれかが成立することを示せ。

(a) $T = \forall\overline{X}.X_i$、ただし $X_i \in \overline{X}$

(b) $T = \forall\overline{X}_1\overline{X}_2.T_1 \to T_2$、ただしある型の列 $\overline{A}$ と $\overline{B}$ について、$|\overline{A}| = |\overline{X}_1\overline{X}_2|$ かつ $|\overline{B}| = |\overline{X}_1|$ であり、$[\overline{X}_1\overline{X}_2 \mapsto \overline{A}]T_1 = [\overline{X}_1 \mapsto \overline{B}](\forall\overline{Z}.T_1 \to T_2)$

(5) *erase*(s) = λx.m かつ $\Gamma \vdash$ s : S ならば、ある $\overline{X}$、S_1、S_2 があり、S は $\forall\overline{X}.S_1 \to S_2$ という形になることを示せ。

(6) 型 T の**最左の葉**を以下のように定義する。

$$\begin{aligned} \textit{leftmost-leaf}(X) &= X \\ \textit{leftmost-leaf}(S \to T) &= \textit{leftmost-leaf}(S) \\ \textit{leftmost-leaf}(\forall X.S) &= \textit{leftmost-leaf}(S) \end{aligned}$$

$[\overline{X}_1\overline{X}_2 \mapsto \overline{A}](\forall\overline{Y}.T_1) = [\overline{X}_1 \mapsto \overline{B}](\forall\overline{Z}.(\forall\overline{Y}.T_1) \to T_2)$ ならば、ある $X_i \in \overline{X}_1\overline{X}_2$ に対して *leftmost-leaf*(T_1) = X_i でなければならないことを示せ。

(7) omega が System F で型付け不能であることを示せ。 □

23.7 型消去と評価順序

図 23-1 に示した System F の操作的意味論は**型渡し意味論**である。つまり、多相関数が型引数に出会ったとき、関数の本体に型が実際に代入される。第 25 章の System F の ML 実装ではまさにこのようにしている。

System F を基礎としたプログラミング言語の、より現実的なインタプリタやコンパイラでは、実行時に型をこのように扱うのは大きなコストとなる。しかも、実行時に型注釈が大した役割を持たないことはすぐにわかる。実行時に型に基づいて何かが決定されることはないということである。正しく型付けされたプログラムをもとに、型注釈を好きに書き換えて、全く同じように振る舞うプログラムを作れる。このような理由で、多相的な言語の多くは**型消去意味論**を採用している。この意味論では、型検査の後で、型をすべて消した型無し項を解釈したり機械語へコンパイルしたりする[†5]。

しかし、書き換え可能な参照や例外など、副作用のある機能を持つ本格的なプログラミング言語では、型消去関数を定義する際、23.6 節で示した完全な型消去関数より少し慎重にやる必要がある。例えば、System F に例外を投げるプリミティブ error（14.1 節）を追加したとする。項

```
let f = (λX.error) in 0;
```

は評価したら 0 になる。λX.error は構文的な値であり、その本体の error は決して評価されないためである。一方、型を消去した項

[†5] **キャスト**のような機能（151 ページ）が存在する言語では、型渡し実装をせざるをえない。そのような言語で高性能な実装をする典型的な方法は、実行時にわずかな型情報だけ保持するようにすることである。例えば型を渡すのを、それらが実際に使われる可能性のある多相関数のみにする、などである。

```
let f = error in 0;
```

は評価されると例外を投げてしまう[†6]。このことから、実は型抽象は意味論的に重要な役割を果たすことがわかる。値呼び評価戦略では型抽象が評価を止めるので、副作用のあるプリミティブの評価を遅延・抑制するからである。

この食い違いは、値呼び評価を考慮した新しい型消去法を導入すれば解決できる。つまり、型抽象を**項**抽象に変えることで消せばよい。

$$
\begin{array}{lcl}
erase_v(\mathtt{x}) & = & \mathtt{x} \\
erase_v(\lambda\mathtt{x{:}T_1.\ t_2}) & = & \lambda\mathtt{x}.\ erase_v(\mathtt{t_2}) \\
erase_v(\mathtt{t_1\ t_2}) & = & erase_v(\mathtt{t_1})\ erase_v(\mathtt{t_2}) \\
erase_v(\lambda\mathtt{X.\ t_2}) & = & \lambda_.\ erase_v(\mathtt{t_2}) \\
erase_v(\mathtt{t_1\ [T_2]}) & = & erase_v(\mathtt{t_1})\ \mathtt{dummyv}
\end{array}
$$

ここで、dummyv は unit など、任意の型無し値である[†7]。この新しい型消去関数が適切であることは、それが型無し評価と「可換」であるという観察からわかる。つまり、型消去と評価をどちらから行ってもよいということである。

定理 23.7.2. $erase_v(\mathtt{t}) = \mathtt{u}$ ならば、(1) それぞれの評価関係によって t と u が共に正規形であるか、または (2) $\mathtt{t} \longrightarrow \mathtt{t'}$ かつ $\mathtt{u} \longrightarrow \mathtt{u'}$ で、$erase_v(\mathtt{t'}) = \mathtt{u'}$ を満たす。 □

23.8 制限された System F

優雅で強力な System F は、多相性の理論研究で中心的な役割を果たしてきた。しかし言語設計の観点では、能力のすべてが生かされることが滅多にない機能を得る代償に型再構築を捨てるのは、厳しすぎると考えられることがあった。このことから、System F の一部を制限して型再構築の問題を扱えるようにする提案は多数なされてきた。

その中で最も普及したものは ML の let 多相（22.7 節）である。これはしばしば**冠頭多相**ともいわれる。型変数は量化子のない型（**単型**）の上だけを動き、量化された型（**多型**または**型スキーム**）が矢印の左側に現れることを許さない、という制限を System F に課したものとみなせるためである。ML における let の特殊な役割のせいで、この対応を正確に述べるのはいささかややこしい。詳しくは Jim [238] を見よ。

System F の制限としては、上述したもの以外に**ランク 2 多相**もよく研究されている。これは Leivant [277] によって導入され、さらに数多くの人々によって研究された（Jim [238; 239] を見よ）。型を木として見たとき、その根から ∀ 量化子へのパスで、二回以上矢印の左側を通るものがないとき、その型はランク 2 であるといわれる。例えば、(∀X.X→X)→Nat はランク 2 で、Nat→Nat や Nat→(∀X.X→X)→Nat→Nat もランク 2 だが、((∀X.X→X)→Nat)→Nat はランク 2 ではない。ランク 2 の体系では、すべての型がランク 2 に制限される。この体系は、より多くの型無しラムダ項に型を割り当てられるという意味で、冠頭多相（ML 多相）より少し強力である。

Kfoury and Tiuryn [257] は、System F をランク 2 に制限した体系における型再構築の複雑さが、

[†6] これは、22.7 節で説明した、参照と ML 式の let 多相の健全でない組み合わせで起きる問題と関係している。その例の let 本体の一**般化**は、ここでは明示的な型抽象に対応する。

演習 23.7.1. [★★] 263 ページの健全でない例を、参照で拡張された System F（図 13-1）に翻訳せよ。 □

[†7] 一方 22.7 節では、副作用があるときでも ML 式の型再構築の健全性を回復するために**値制限**を導入した。値制限は型抽象を消す（型変数の一般化は本質的に型抽象の消去の反対である）が、健全性は保証している。推論された型抽象が項抽象や、その他構文的な値の構築子に隣接して現れる場合のみ一般化を許しており、これらも評価を止めるからである。

ML における型再構築の複雑さと同じ（つまり DEXPTIME 完全）であると証明した。Kfoury and Wells [255] は、ランク 2 の体系に対する直接的な型再構築アルゴリズムをはじめて与えた。さらに、ランク 3 以上の System F に対する型再構築が決定不能であることを示した。

ランク 2 の制限は、量化子以外の強力な型構築子にも適用できる。例えば**交差型**（15.7 節を見よ）では、交差が二つ以上の矢印の左側に現れる型を排除することでランク 2 に制限できる（Kfoury, Mairson, Turbak, and Wells [256]）。System F をランク 2 に制限したものは、一階交差型システムと強く関係している。実際、Jim [238] はこれらが全く同じ型無し項を型付けできることを示した。

23.9 パラメータ性

273 ページの、Church ブール値の型 CBool の定義のやり方を思い出そう。

```
CBool = ∀X.X→X→X;
```

定数 tru と fls は以下のように定義した。

```
  tru = λX. λt:X. λf:X. t;
▸ tru : CBool

  fls = λX. λt:X. λf:X. f;
▸ fls : CBool
```

型 CBool が与えられたら、ただ型の構造を見るだけで、機械的に tru と fls の定義が書ける。CBool は ∀ で始まるから、型 CBool の任意の値は型抽象でなければならない。よって tru と fls はどちらも λX で始まる。そして CBool の本体は矢印型 X→X→X だから、この型の値はすべて型 X の引数を二つ受け取る。つまり、tru と fls の本体はどちらも λt:X.λf:X で始まる。最後に、CBool の結果の型は X なので、型 CBool の任意の値は型 X の要素を返さなければならない。しかし X はパラメータだから、この型の値で返せるのは束縛変数 t か f しかない。この型の値を自分で作ったり取得したりする手段はないのだ。つまり、tru と fls は本質的に型 CBool の**ただ二つの**要素ということだ。厳密に言うと、CBool は (λb:CBool.b) tru など他の項も含むが、どれも tru か fls と同じように振る舞うことは直観的に明らかである。

この観察は、**パラメータ性**と呼ばれる強力な原理の単純な帰結である。この原理は、多相的なプログラムの振る舞いの一様性を定式化したものである。パラメータ性は Reynolds [408; 413] によって導入され、Reynolds [414]、Reynolds and Plotkin [421]、Bainbridge et al. [35]、Ma [292]、Mitchell [319]、Mitchell and Meyer [324]、Hasegawa [200]、Pitts [377; 378; 379]、Abadi, Cardelli, and Curien [5]、Plotkin and Abadi [382]、Plotkin, Abadi, and Cardelli [383]、Wadler [479; 483]、他によって、関連する概念と共に深く研究されてきた。解説的な導入は Wadler [479] を見よ。

23.10 非可述性

System F の多相性はしばしば**非可述的**であるといわれる。一般に、（集合、型、その他の）定義は、定義されているそのもの自身が定義域に含まれるような量化子を含むとき、「非可述的」であるといわれる。例えば System F では、型 T = ∀X.X→X の型変数 X は任意の型の上を動く。これには T 自身も含まれる（これは、例えば、型 T に型 T の項を具体化して、T から T への関数を得られるようにするためだ）。一方 ML に見られる多相性はしばしば**可述的**である（または**階層化**されている）といわれる。これは型変数の値域が単型に制限されていて、量化子を含まないためである。

「可述的」「非可述的」の用語は論理学に起因する。Quine [393] はこの歴史をわかりやすくまとめている。

Henri Poincaré との交流の中で、...Russell は、(Russell の) パラドックスの原因が、Russell が悪循環の誤謬と呼ぶものではないかと考えた。この「誤謬」は、定義しようとしている類そのものを含む類の範囲を直接的または間接的に参照する所属条件によって類を定義しようとするところにあった。例えば Russell のパラドックスの背後にある所属条件は、x が x を含まない、つまり、自己を含まないということだった。所属条件の x を、わざわざ、この所属条件によって定義しようとしている類そのものにしているところにこのパラドックスは起因している。Russell と Poincaré はこのような所属条件を**非可述的**と呼び、類を定義する手段として認めないことにした。こうして、Russell のパラドックスやその他集合論のパラドックスは取り除かれた...

用語法の問題であるが、何ゆえに「可述的」「非可述的」になったのだろうか。Russell の言によれば、類と所属条件に関する、「すべての述語は類を定義する」という言葉はもう使いものにならなくなった。そこで Russell は、そういう所属条件がもはや類を定義しているものには見えないよう、述語という肩書きを奪うことでつじつま合わせをする。つまり「可述的」は、特に階層的なアプローチや、進行的な構成のメタファを暗示するものではなかった。単に、類を定義するもの、つまり述語 (「可述的」なもの) としてどのような所属条件を受け入れるべきか、ということについて Russell と Poincaré が行った特定の提案にすぎなかった。しかしすぐに本末転倒になった。今日では可述的集合論は構成的集合論であり、非可述的定義は前の段落できっちり説明した通りで、どの所属条件が類を定義していると考えるかには関係ない。

23.11 注記

System F に関するさらなる文献は、Reynolds による導入記事 [417] や "*Theories of Programming Languages*" [420] から辿ることができる。

第 24 章

存在型

本章では主に、System F（図 23-1）に存在型（図 24-1）を加えた体系を扱う。この例でもレコード（図 11-7）や数（図 8-2）を使う。本章に対応する OCaml 実装は `fullpoly` である。

型システムにおける全称量化子の役割を学んだので（第 23 章）、**存在**量化子もプログラミング言語で有用だろうかと考えるのは自然なことである。そしてこの期待は正しい。存在量化子はデータ抽象や情報隠蔽のための洗練された基礎となる。

24.1 動機

基本的に、存在型は全称型以上に複雑なものではない（実際 24.3 節で、存在型が全称型で直接的に表現できることを説明する）。しかし存在型の導入・除去の形式は、全称型で使った単純な型抽象・適用よりも構文的に少し重苦しい。最初はちょっとわかりにくいと感じる人もいる。そういう局面を乗り越えるには、次の二つの直観が役立つだろう。

第 23 章の全称型には二つの異なる見方がある。**論理学的直観**では、型 ∀X.T の要素は、任意の S について型 [X ↦ S]T の値である。この直観は型を消去して振る舞いを考える方法に対応する。例えば、多相的な恒等関数 λX.λx:X.x から型を消すと、型無しの恒等関数 λx.x になる。これは任意の型 S の引数から、同じ型の返り値へ写す関数である。一方、**操作的直観**では、∀X.T の要素は、型 S から、[X ↦ S]T という型に特化した項へ写す関数である。この直観は第 23 章での System F の定義に対応し、型適用の簡約を実際の計算ステップとみなす。

同様に、存在型 {∃X,T} にも二つの考え方がある。**論理学的直観**では、{∃X,T} の要素は、**ある**型 S について型 [X ↦ S]T の値である。一方**操作的直観**では、{∃X,T} の要素は、型 S と、[X ↦ S]T 型の項 t との二つ組（{*S,t} と書く）である。

本章で扱う存在型では、操作的な考え方を強調していく。こちらのほうが、プログラミング言語に見られるモジュールや抽象データ型と、存在型との間の強い類推を与えるからである。本書ではこの類推を踏まえ、存在型の具象構文として {∃X,T} と書く。波括弧によって、存在型の値が組の一種であることを強調している。（より標準的な表記は ∃X.T である。）

存在型を理解するには知らなければならないことが二つある。存在型の要素を**作る**（9.4 節の用語では**導入する**）方法と、計算の中でその値を**使う**（**除去する**）方法である。

存在型を持つ値は、型を項と組み合わせることで導入される。これを {*S,t} と書く[†1]。有用な具体的直観としては、型 {∃X,T} の値 {*S,t} を、（隠蔽された）型が一つ、項が一つからなる**パッケージ**や**モジュール**の簡単な形と考えられる[†2]。この型 S はよく、**隠蔽表現型**と呼ばれる。あるいは（論理体系とのつながり（9.4 節を参照）を強調するため）パッケージの**証人型**と呼ばれることもある。例えば、パッケージ p = {*Nat, {a=5, f=λx:Nat. succ(x)}} は、存在型 {∃X, {a:X,

[†1] 従来の項の組（11.7 節）と間違えないよう、こちらの組では型に * の記号を付ける。存在型の導入には、pack X=S with t という表記もよく使われる。

[†2] このモジュールを複数の型や値の要素に一般化することは誰でも思いつくことだろう。しかし表記を扱いやすくするため、ここでは一つだけにとどめておこう。型の要素を複数持たせたければ、単一の存在型を入れ子にすれば同じ効果が得られる。項の要素を複数持たせたければ、右側の要素として組やレコードを使えばよい。

$$\{*S_1, *S_2, t_1, t_2\} \stackrel{\text{def}}{=} \{*S_1, \{*S_2, \{t_1, t_2\}\}\}$$

f:X→X}} を持つ。p の型の要素は Nat である。値の要素は、ある型 X（つまり Nat）について、型 X のフィールド a と型 X→X のフィールド f を持つレコードである。

パッケージ p は、同時に {∃X, {a:X, f:X→Nat}} という型も持つ。右側の要素が、ある型 X（つまり Nat）について、それぞれ型 X、X→Nat のフィールド a と f を持つレコードだからである。この例から、与えられたパッケージがどの存在型に属すかについて、型検査器が自動的に判断することは一般的に不可能だとわかる。したがって、プログラマがどれを意図しているか指示しなければならない。これを行う簡単な方法は、単にすべてのパッケージに意図している型を注釈として明示することである。よって存在型の導入の完全な構文は、

```
  p = {*Nat, {a=5, f=λx:Nat. succ(x)}} as {∃X, {a:X, f:X→X}};
▸ p : {∃X, {a:X,f:X→X}}
```

もしくは、別の型にすると、

```
  p1 = {*Nat, {a=5, f=λx:Nat. succ(x)}} as {∃X, {a:X, f:X→Nat}};
▸ p1 : {∃X, {a:X,f:X→Nat}}
```

のようになる。この as による型注釈は 11.4 節で導入した型指定の構文に似ている。前回の構文では意図する型の注釈を**任意の**項に付けることができた。今回は本質的には、パッケージの具象構文の一部として単一の型指定を組み込んでいる。存在型の導入の型付け規則を以下に示す。

$$\frac{\Gamma \vdash t_2 : [X \mapsto U]T_2}{\Gamma \vdash \{*U, t_2\} \text{ as } \{\exists X, T_2\} : \{\exists X, T_2\}} \qquad \text{(T-PACK)}$$

この規則で一つ気がつくのは、**同じ**存在型に、**異なる**隠蔽表現型のパッケージが属することである。例えば

```
  p2 = {*Nat, 0} as {∃X,X};
▸ p2 : {∃X, X}

  p3 = {*Bool, true} as {∃X,X};
▸ p3 : {∃X, X}
```

あるいは、より実用的な例としては、

```
  p4 = {*Nat, {a=0, f=λx:Nat. succ(x)}} as {∃X, {a:X, f:X→Nat}};
▸ p4 : {∃X, {a:X,f:X→Nat}}

  p5 = {*Bool, {a=true, f=λx:Bool. 0}} as {∃X, {a:X, f:X→Nat}};
▸ p5 : {∃X, {a:X,f:X→Nat}}
```

である。

演習 24.1.1. [★]　同じ題材を変形したものを三つ挙げる。

```
  p6 = {*Nat, {a=0, f=λx:Nat. succ(x)}} as {∃X, {a:X, f:X→X}};
▸ p6 : {∃X, {a:X,f:X→X}}

  p7 = {*Nat, {a=0, f=λx:Nat. succ(x)}} as {∃X, {a:X, f:Nat→X}};
▸ p7 : {∃X, {a:X,f:Nat→X}}

  p8 = {*Nat, {a=0, f=λx:Nat. succ(x)}} as {∃X, {a:Nat, f:Nat→Nat}};
▸ p8 : {∃X, {a:Nat,f:Nat→Nat}}
```

これらは、どのような点が、p4 や p5 よりも有用でないか。 □

→ ∀ ∃ System F（図 23-1）の拡張

新しい構文形式

t	::=	...	項：
		{*T,t} as T	パッケージ化
		let {X,x}=t in t	アンパッケージ化
v	::=	...	値：
		{*T,v} as T	パッケージの値
T	::=	...	型：
		{∃X,T}	存在型

新しい評価規則 $\boxed{t \longrightarrow t'}$

$$\begin{array}{l}\texttt{let \{X,x\}=(\{*}T_{11}\texttt{,}v_{12}\texttt{\} as }T_1\texttt{) in }t_2\\ \quad\longrightarrow [X\mapsto T_{11}][x\mapsto v_{12}]t_2\end{array}\qquad\text{(E-UNPACKPACK)}$$

$$\frac{t_{12} \longrightarrow t'_{12}}{\texttt{\{*}T_{11}\texttt{,}t_{12}\texttt{\} as }T_1 \longrightarrow \texttt{\{*}T_{11}\texttt{,}t'_{12}\texttt{\} as }T_1}\qquad\text{(E-PACK)}$$

$$\frac{t_1 \longrightarrow t'_1}{\texttt{let \{X,x\}=}t_1\texttt{ in }t_2 \longrightarrow \texttt{let \{X,x\}=}t'_1\texttt{ in }t_2}\qquad\text{(E-UNPACK)}$$

新しい型付け規則 $\boxed{\Gamma \vdash t : T}$

$$\frac{\Gamma \vdash t_2 : [X\mapsto U]T_2}{\Gamma \vdash \texttt{\{*}U\texttt{,}t_2\texttt{\} as \{}\exists X\texttt{,}T_2\texttt{\}} : \texttt{\{}\exists X\texttt{,}T_2\texttt{\}}}\qquad\text{(T-PACK)}$$

$$\frac{\Gamma \vdash t_1 : \texttt{\{}\exists X\texttt{,}T_{12}\texttt{\}} \qquad \Gamma, X, x\texttt{:}T_{12} \vdash t_2 : T_2}{\Gamma \vdash \texttt{let \{X,x\}=}t_1\texttt{ in }t_2 : T_2}\qquad\text{(T-UNPACK)}$$

▲図 24-1. 存在型

このようなモジュールとの類推から、存在型の除去構文についての有用な直観も得られる。存在型パッケージがモジュールと対応するなら、パッケージの除去は open や import ディレクティブのようなものである。このディレクティブを使うと、プログラムの他の場所でモジュールの要素を使えるようになるが、モジュールが持つ型の要素の正体は抽象的なままである。この直観はパターンマッチによる束縛の一種として実現できる。

$$\frac{\Gamma \vdash t_1 : \texttt{\{}\exists X\texttt{,}T_{12}\texttt{\}} \qquad \Gamma, X, x\texttt{:}T_{12} \vdash t_2 : T_2}{\Gamma \vdash \texttt{let \{X,x\}=}t_1\texttt{ in }t_2 : T_2}\qquad\text{(T-UNPACK)}$$

つまり、t_1 が存在型パッケージを返す式ならば、その型と項の要素をパターン変数 X と x に束縛し、それらを t_2 の計算で使うことができる。（存在型の除去には、open t_1 as {X,x} in t_2 という具象構文も一般的に用いられる。）

例えば、先に定義した、型 {∃X, {a:X, f:X→Nat}} のパッケージ p4 を考える。以下の除去の式は、

```
let {X,x}=p4 in (x.f x.a);
▸ 1 : Nat
```

p4 を開き、パッケージ本体の二つのフィールド（x.f と x.a）を用いて結果の数を計算する。除去形式の本体で型変数 X を使うこともできる。

```
let {X,x}=p4 in (λy:X. x.f y) x.a;
▸ 1 : Nat
```

パッケージ本体の型検査の際、パッケージの表現型は抽象的なものとして扱われる。つまり、x に関して可能な演算は、その「抽象型」{a:X,f:X→Nat} が保証しているものだけということであ

る。特に、x.a を具体的な数として扱うことは許されていない。

```
  let {X,x}=p4 in succ(x.a);
▶ Error: argument of succ is not a number
```

この制限は理にかなっている。前に説明した通り、p4 と同じ存在型のパッケージが自身の表現型としてNatやBool（やその他）を使う可能性があるからである。

他にも、存在型の除去構文の型検査で失敗してしまう、もっと微妙な場合がある。規則 T-UNPACK において型変数Xは、t_2 の型を計算する文脈には現れるが、この規則の結論の文脈には現れ**ない**。つまり、Xの自由な出現は結論のスコープ外であるから、結果の型 T_2 はXを自由に含むことができないということだ。

```
  let {X,x}=p4 in x.a;
▶ Error: Scoping error!
```

このことについては 25.5 節で詳しく議論する。

存在型の計算規則は単純である。

$$\begin{array}{l}\texttt{let }\{X,x\}=(\{*T_{11},v_{12}\}\texttt{ as }T_1)\texttt{ in }t_2\\ \qquad\longrightarrow [X\mapsto T_{11}][x\mapsto v_{12}]t_2\end{array}\qquad\text{(E-UNPACKPACK)}$$

let の最初の部分式が具体的なパッケージに簡約済みであるとき、本体 t_2 内の変数Xとxにパッケージの要素を代入できる。モジュールとの類推という観点で考えると、この規則は**リンク**のステップとみなせる。すなわち、別途コンパイルされるモジュールの中身を参照する記号的な名前（Xとx）を、そのモジュールの実体に置き換えるステップである。

型変数Xはこの規則で代入によって消えるので、結果的にプログラムは、パッケージの内部へ具体的にアクセスできる。これは単に、今まで何度も見てきた現象の別の例にすぎない。計算が進むにつれて、式は「より型付けできる」のである。すなわち、正しく型付けされていなかった式も、正しく型付けされた式に簡約される場合がある。

存在型を加えた System F の拡張を定義する規則を図 24-1 に要約する。

24.2 存在型によるデータ抽象

本書の導入の章（4ページ）で論じたことだが、型システムは 2+true のような小規模なプログラミングのエラーを検出する役割だけではなく、大規模なプログラミングにおいても恐ろしいほど効果がある。特に、言語組み込みの抽象を守らせる（つまり、プログラムから計算機を保護する）だけでなく、**プログラマが定義した抽象**を守らせる（つまり、プログラムの部品同士をお互いから保護する）ことにも型は役立つ[†3]。本節では二つの異なる抽象化の流儀（古典的な**抽象データ型**と、**オブジェクト**）について、存在型を共通の土台として議論する。

第18章で登場したオブジェクト表現とは異なり、この節で挙げる例はすべて**純粋関数的な**プログラムである。これを選択したのは解説のためにすぎない。というのも、モジュール性と抽象のための機構は、定義しようとしている抽象が状態を持つかどうかとはほぼ完全に直交しているのだ（演習 24.2.2 および演習 24.2.4 では、本文での純粋関数的な例の一部を、状態を使って書き換える。このことから前述の直交性がわかる）。ここで純粋関数的な例を選ぶ理由は、(1) ここに挙げる例が、より単純で無駄のない形式化でも有効であるとわかること、そして (2) 純粋関数的なプログラムに取り組むと、型付けの問題がより面白くなることがあること（そしてその解決方法もより理解しやす

[†3] 公平のために補足する。プログラマが定義した抽象を保護する方法が型しかないわけではない。型無し言語でも、関数閉包、オブジェクト、専用の構文（MzScheme の **unit**（Flatt and Felleisen [157]）など）を使って同じようなことができる。

くなること）である。なぜなら手続き型プログラミングでは、書き換え可能な変数が「サイドチャネル」になるからである。つまり、その変数を使って、プログラム中の離れた場所が直接通信できてしまう。純粋関数的なプログラムでは、プログラムの異なる部分がやり取りする情報はすべて関数の引数か返り値を通じて受け渡されると決まっている。これなら型システムから「見える」。以上は特にオブジェクトの場合にいえることで、重要な機能の幾つか（部分型付けと継承）は第 32 章まで扱わないことにする。第 32 章では、より強力な型理論的機構を利用できるようになっている。

抽象データ型

従来の**抽象データ型**（**ADT**）は、(1) 型名 A、(2) 具象表現型 T、(3) T 型の値の作成・問い合わせ・操作をするための演算の実装、(4) 表現と演算を取り囲む**抽象境界**、の四つからなる。この境界の内部では、型の要素は具体的に（すなわち型 T として）見える。外部では、これらは抽象的に型 A として見える。型 A の値は、データ構造に格納するなどして他に渡される可能性があるが、それでも直接覗き込まれたり変えられたりするようなことはない。A に許されている演算は、ADT が提供しているものだけである。

例を挙げる。以下は純粋関数的なカウンタの抽象データ型の宣言を、Ada（U.S. Dept. of Defense [472]）や CLU（Liskov et al. [288]）風の擬似コードとして書いたものである。

```
ADT counter =
  type Counter
  representation Nat
  signature
     new : Counter,
     get : Counter→Nat,
     inc : Counter→Counter;
  operations
     new = 1,
     get = λi:Nat. i,
     inc = λi:Nat. succ(i);

counter.get (counter.inc counter.new);
```

抽象型の名前は Counter である。その具象表現は Nat である。new は単に定数 1 であり、get は恒等関数であり、inc 操作は後者関数である。このように演算の実装は、Counter オブジェクトを具体的に Nat として扱っている。signature 節では、演算の具体的な型が含む Nat を幾つか Counter に置き換えることで、演算が外部からどのように使われるかを指定している。抽象境界は、ADT キーワードから最後のセミコロンまでとなる。プログラムの残りの部分（つまり最後の行）では、Counter と Nat の対応関係は消されている。そのため定数 counter.new は、counter.get か counter.inc の引数として使うことしかできない。

この擬似コードは存在型を伴う計算体系に、ほぼ記号の一対一対応により変換できる。まず ADT の内部実装を含む存在型パッケージを作る。

```
counterADT =
   {*Nat,
    {new = 1,
     get = λi:Nat. i,
     inc = λi:Nat. succ(i)}}
 as {∃Counter,
     {new: Counter,
      get: Counter→Nat,
      inc: Counter→Counter}};
▶ counterADT : {∃Counter,
                {new:Counter,get:Counter→Nat,inc:Counter→Counter}}
```

そしてこのパッケージを開く。パッケージの隠蔽表現型のプレースホルダーとして型変数 Counter が導入され、さらに演算へのアクセスを提供する項変数 counter が導入される。

```
let {Counter,counter} = counterADT in
counter.get (counter.inc counter.new);
▶ 2 : Nat
```

糖衣構文を使った擬似コードと比べると、存在型を使ったものはちょっと目にやさしくないが、この二つは構造としては等価である。

一般に、存在型パッケージを開く let の本体には、プログラムの残り全体を置く。

```
let {Counter,counter} = <カウンタパッケージ> in
<プログラムの残り>
```

この残りの部分では、型名 Counter を言語組み込みの基本型のように使える。カウンタを操作する関数は以下のように定義できる。

```
let {Counter,counter}=counterADT in
let add3 = λc:Counter. counter.inc (counter.inc (counter.inc c)) in
counter.get (add3 counter.new);
▶ 4 : Nat
```

表現の一部にカウンタを含むような、別の抽象データ型を定義することもできる。例えば以下のプログラムは、(あまり効率的ではないが) カウンタを表現型として使ったフリップフロップの ADT を定義している。

```
let {Counter,counter} = counterADT in

let {FlipFlop,flipflop} =
     {*Counter,
      {new    = counter.new,
       read   = λc:Counter. iseven (counter.get c),
       toggle = λc:Counter. counter.inc c,
       reset  = λc:Counter. counter.new}}
   as {∃FlipFlop,
       {new:    FlipFlop, read: FlipFlop→Bool,
        toggle: FlipFlop→FlipFlop, reset: FlipFlop→FlipFlop}} in

flipflop.read (flipflop.toggle (flipflop.toggle flipflop.new));
▶ false : Bool
```

以上のように、大きなプログラムは、多数の ADT 宣言を並べた形に分解できる。各 ADT 宣言は、それより前の ADT 宣言が提供している型と演算を用いて、新たな型や演算を実装し、正しく定義さ

れたきれいな抽象としてパッケージ化し、後の ADT に提供する。

ここで行った一種の情報隠蔽は、**表現非依存性**という重要な性質を持っている。つまり、プログラム全体の型安全性が保たれるという完全な信頼の下で、Counter の ADT を別実装に置き換えられる。例えば、内部表現として単なる Nat ではなく、Nat を含むレコードに置き換えることが考えられる。

```
counterADT =
   {*{x:Nat},
    {new = {x=1},
     get = λi:{x:Nat}. i.x,
     inc = λi:{x:Nat}. {x=succ(i.x)}}}
  as {∃Counter,
     {new: Counter, get: Counter→Nat, inc: Counter→Counter}};
▸ counterADT : {∃Counter,
                 {new:Counter,get:Counter→Nat,inc:Counter→Counter}}
```

このようにできるのは、プログラムの残りの部分が get と inc 以外の方法で Counter のインスタンスにアクセスすることはできない、ということが保証されているからである。

抽象データ型に基づいたプログラミングスタイルが大規模システムの堅牢性・保守性の向上に非常に効果的であるということは、経験的に知られていることである。これには幾つか理由がある。一つは、このスタイルだとプログラムへの変更範囲が限定されることである。先ほど説明したように、具象表現型やその演算の実装を変えることで、ADT の実装を他のものに差し替えられるのだが、このときプログラムの残りの部分には影響がない。なぜなら、プログラムの残り部分が ADT の内部表現に依存できないということを、存在型パッケージの型付け規則が担保しているからである。二つ目の理由は、プログラマが ADT のシグネチャをできるだけ小さくすることで、プログラムの部分間の依存性を限定するよう ADT が後押しすることである。最後の、おそらく最も重要な理由は、演算のシグネチャを明示することで、プログラマに**抽象化の設計**を考えさせることである。

演習 24.2.1. [推奨, ★★★] 上の例を参考に、数の**スタック**の抽象データ型を定義せよ。new、push、top（現在一番上にある要素を返す）、pop（一番上の要素を取り除いた新しいスタックを返す）、isempty という操作を備えるものとする。23.4 節で導入した List 型を内部表現として用いよ。スタックを作り、数を幾つか入れて、一番上の要素を取り出す簡単なプログラムを書け。この演習では実際に手を動かしてみるのが一番だ。fullomega 検査器を使い、fullomega ディレクトリのファイル test.f の内容をファイル先頭にコピーして用いよ（List 構築子やそれに関係する操作の定義がある）。 □

演習 24.2.2. [推奨, ★★] 第 13 章で定義した参照セルを用いて、**書き換え可能**なカウンタの ADT を作れ。new を、ただの定数から、Unit 引数を受け取って Counter を返す関数に変更せよ。inc 操作の返り値の型を、Counter から Unit に変更せよ。（fullomega 検査器は存在型に加えて参照を提供する。） □

存在型オブジェクト

前の小節では、「パッケージ化してすぐ開く」というイディオムを使った。これは存在型パッケージを使った ADT 流のプログラミングに特有のものである。パッケージは抽象型とそれに関係する演算を定義し、その直後で各パッケージを開く。これによって抽象型を型変数に束縛し、ADT の演算を抽象的に開示した。本節では、ある種の単純な**オブジェクト**流のデータ抽象が、存在型に基づ

いた別のプログラミングイディオムともみなせること、およびその方法について述べる。第 32 章ではこのオブジェクトモデルをさらに拡張する。

今回も単純なカウンタを題材として用いる。前述した存在型による ADT でも、以前オブジェクトを説明した第 18 章と第 19 章でも用いた題材である。今回も純粋関数的な形式を用いる。つまり、カウンタにメッセージ inc を送ってもその内部状態を直接変更するわけではなく、1 増えた内部状態を持つ**新しい**カウンタオブジェクトを返す。

カウンタオブジェクトは二つの基本要素からなる。数（内部状態）と、その状態を操作する get と inc という二つのメソッドである。この内部状態を調べたり更新したりする唯一の方法が、どちらかのメソッドを使うことであると保証する必要もある。これには、状態とメソッドを存在型でパッケージ化し、状態の型を抽象化してしまえばよい。例えば、値 5 を保持するカウンタオブジェクトは

```
c = {*Nat,
     {state = 5,
      methods = {get = λx:Nat. x,
                 inc = λx:Nat. succ(x)}}}
    as Counter;
```

と書ける。ただし

```
Counter = {∃X, {state:X, methods: {get:X→Nat, inc:X→X}}};
```

である。

カウンタオブジェクトのメソッドを使うには、存在型を開いて、methods の適切な要素を state フィールドに適用する。例えば c の現在の値を得るには、

```
let {X,body} = c in body.methods.get(body.state);
▶ 5 : Nat
```

と書けばよい。より一般的に、任意のカウンタに「get メッセージを送る」小さな関数は以下のように定義できる。

```
sendget = λc:Counter.
            let {X,body} = c in
            body.methods.get(body.state);
▶ sendget : Counter → Nat
```

カウンタオブジェクトの inc メソッドを呼ぶのはもう少しややこしい。get と同じようにするだけだと、型検査器がエラーを出す。

```
let {X,body} = c in body.methods.inc(body.state);
▶ Error: Scoping error!
```

なぜなら、型変数 X が let の本体の型として自由に出現してしまうからである。実際、今書いたものは直観的にも理にかなったものではない。inc メソッドの返り値は内部状態そのものであり、オブジェクトではないからである。型検査を通し、かつ、inc を呼び出したときの動作として期待されることを満たすには、この新しい内部状態を受け取って、カウンタオブジェクトとして再パッケージ化しなければならない。メソッドのレコードと内部状態の型は、元々のオブジェクトと同じものを使えばよい。

```
c1 = let {X,body} = c in
       {*X,
        {state = body.methods.inc(body.state),
         methods = body.methods}}
     as Counter;
```

より一般的に、カウンタに「inc メッセージを送る」関数は、

```
sendinc = λc:Counter.
            let {X,body} = c in
              {*X,
               {state = body.methods.inc(body.state),
                methods = body.methods}}
              as Counter;
▸ sendinc : Counter → Counter
```

と書ける。カウンタに対する、より複雑な操作は、この二つの基本操作を使って実装できる。

```
add3 = λc:Counter. sendinc (sendinc (sendinc c));
▸ add3 : Counter → Counter
```

演習 24.2.3. [推奨, ★★★] 前述した FlipFlop の ADT を参考に、Counter オブジェクトを内部表現の型として使って FlipFlop オブジェクトを実装せよ。 □

演習 24.2.4. [推奨, ★★] fullomega 検査器を使って、演習 24.2.2 に従い、状態を持つ Counter オブジェクトを実装せよ。 □

オブジェクト 対 ADT

前節の例は、オブジェクト指向プログラミングの本格的なモデルに基づいたものではない。第 18 章と第 19 章で説明した多くの機能、つまり部分型付け、クラス、継承、self と super を経由した再帰などがここでは欠けている。これらの機能のモデル化については、型システムに必要な改良を幾つか加えた後で、第 32 章で再び取り上げる。しかし、ここまで議論してきた簡単なオブジェクトと ADT の間でも、興味深い比較がいくらか可能である。

最も粗いレベルでいうと、二つのプログラミングのイディオムは両極端である。ADT でのプログラミングでは、パッケージは作ったらすぐに開く。一方、オブジェクトをモデル化するパッケージは可能な限り閉じられたままである。つまり、メソッドを内部状態に適用するため、パッケージを**開かざるをえない**瞬間まで閉じられたままである。

この違いからは、「カウンタの抽象型」が指し示すものに二つの流儀によって違いがあることがわかる。ADT 流のプログラムでは、add3 関数のようなクライアントコードが操作するカウンタの値は、内部表現型の要素（例えばただの数）である。オブジェクト流のプログラムでは、それぞれのカウンタはパッケージ全体となっていて、数だけでなく get や inc のメソッドの実装も含んでいる。この流儀の違いは型 Counter の定義に反映されている。ADT 流では、型 Counter は let 構文が導入する束縛型変数である。一方オブジェクト流では、型 Counter は

```
{∃X, {state:X, methods: {get:X→Nat,inc:X→X}}}
```

という存在型全体を表す。よって実行時には、カウンタ ADT から生成されたカウンタの値はすべて、内部表現型の要素そのものであり、その内部表現に対するカウンタ演算の実装はただ一つだけ存在する。一方、カウンタオブジェクトは、自身の表現型だけでなく、その表現型に対するメソッドの集合をも持っている。

オブジェクトと ADT はこのように異なるので、実際の利点も対照的になる。オブジェクトに対する明らかな利点を一つ挙げる。オブジェクトごとに内部表現を選択し、演算を持ち運ぶので、一つのプログラムの中で同じオブジェクト型に異なる実装を複数、混ぜて使える。このことは部分型や継承と併せて使うと極めて便利である。オブジェクトの一般的なクラスを一つ定義し、少し（または完全に）異なる表現を持つ詳細化版を複数提供できる。この詳細化されたクラスのインスタンスは、同じ一般型を共有しているので、同じ一般的なコードで扱ったり、同じリストに一緒に格納したりできる。

例えばユーザインターフェイスのライブラリでは、一般的な `Window` クラスを定義し、さらに `TextWindow`、`ContainerWindow`、`ScrollableWindow`、`TitledWindow`、`DialogBox` のようなサブクラスを定義するだろう。これらのサブクラスは、それぞれ固有のインスタンス変数を持つ（例えば `TextWindow` は、現在の内容を表す `String` のインスタンス変数を持つ。一方 `ContainerWindow` は、`Window` オブジェクトのリストを持つかもしれない）。また、`repaint` や `handleMouseEvent` などの操作は、サブクラスごとに特化した実装を用意する。一方、`Window` を ADT として定義すると、柔軟性に欠けた構造になってしまう。`Window` の具象表現型はバリアント型（11.10 節）を含む必要があるだろう。一つ一つの特定の種類のウィンドウに一つの構築子が対応し、その種類のウィンドウに関連する特有のデータを持ち運ぶ。`repaint` のような操作は、バリアントに `case` 構文を使って、それぞれ適切なコードを実行する。もしウィンドウの種類がたくさんあったら、`Window` の ADT は一枚岩のような宣言なので、すぐ肥大化して手に負えなくなるだろう。

他にオブジェクトと ADT の実用上の差として、**二項演算**に関するものがある。二項演算とは、同じ抽象型の引数を二つ（以上）受け取る演算のことである。この話をきちんと議論するには、二項演算を二つに分類する必要がある。

- 一部の二項演算は、二つの抽象値に対して、公開されている演算を使うだけで完全に実装できる。例えばカウンタの等価判定演算を実装するには、（`get` を使って）それぞれの現在値を調べ、結果を比較するだけでよい。つまり `equal` 演算は、カウンタの具象表現を守る抽象境界の外側で定義できるということである。このような演算を**弱二項演算**と呼ぶ。
- 二項演算には、それぞれの抽象値の内部表現にアクセスする特権がないと実装できないものもある。例えば、数の集合を表現する抽象を実装するとしよう。アルゴリズムの教科書をあさり、集合の具象表現として、とある複雑な不変条件に従うラベル付きの木を用いることにしたとする。二つの集合の `union` 演算を効率的に実装するには、それぞれの集合を具体的に、つまり木として扱う必要がある。しかし、集合の抽象に対する公開インターフェイスとして、どこからでも具象表現を見えるようにしたいわけがない。そこで、普通のクライアントコードは利用できない特権を `union` だけに与え、両引数へのアクセスを許す必要がある。つまり `union` 演算は、抽象境界の内側に存在しなければならない。このような演算を**強二項演算**と呼ぶ。

弱二項演算は、オブジェクト流でも ADT 流でも簡単なケースである。この演算は、抽象境界の中に置こうが外に置こうが、大した差はないからである。もし外に置く場合は、単にトップレベルの関数として定義すればよいだろう（オブジェクトか ADT の値か、適切なものを受け取る関数になる）。弱二項演算を ADT の中に置く場合は今までと全く同じである（実際には必要なくとも、演算は引数の表現に具体的にアクセスできる）。オブジェクトの中に置く場合は、オブジェクトの型が再帰的になるので、少しだけ大変である[†4]。

[†4] オブジェクト型のこの種の再帰は、継承がらみでややこしい。Bruce et al. [61] を見よ。

```
EqCounter = {∃X, {state:X, methods: {get:X→Nat, inc:X→X,
                                     eq:X→EqCounter→Bool}}}
```

一方、今回のモデルでは、強二項演算をオブジェクトのメソッドとして表現することはできない。強二項演算の**型**だけであれば、単に先の弱二項演算と同じように表現できる。

```
NatSet = {∃X, {state:X, methods: {empty:X, singleton:Nat→X,
                                  member:X→Nat→Bool,
                                  union:X→NatSet→X}}}
```

だがこの型のオブジェクトをちゃんと**実装**する方法がない。union 演算の第二引数についてわかるのは、それが NatSet の演算を提供しているということだけである。提供されている演算の中に、集合の要素が何なのかを知る手段がないので、和集合を計算することはできない。

演習 24.2.5. [★] 以下の型も使えない。なぜだろうか。

```
NatSet = {∃X, {state:X, methods: {empty:X, singleton:Nat→X,
                                  member:X→Nat→Bool,
                                  union:X→X→X}}}
```
□

要するに、一つの ADT は一つの表現しか持たないので、二項演算を直にサポートする一方、オブジェクトには複数の表現がありうるので、有用な柔軟さが得られる引き換えに二項演算を諦めている。これは一長一短であり、どちらかがどちらかより優れているというものではない。

この議論について、一つ断っておくべきことがある。以上の比較は、本章の最初で示した、単純で「純粋主義的」なオブジェクトモデルにおいて有効なものである。C++ や Java など主流のオブジェクト指向言語のクラスは、ある程度の強二項演算メソッドが記述可能なように設計されている。これは本章で見てきた純粋なオブジェクトと純粋な ADT の折衷案と考えるのが実際にはベストだろう。これらの言語においてオブジェクトの型は、そのオブジェクトが属するクラスの名前そのものである。この型は、他のクラスの名前とは異なるものとみなされる。たとえ全く同じ演算の集合を提供しているクラスであっても、別の型となる（19.3 節参照）。つまりこれらの言語では、オブジェクトの型は、それに対応するクラス宣言で定義された実装を**ただ一つ**持つ。また、サブクラスは、スーパークラスから継承されたインスタンス変数に、新たなインスタンス変数を追加することしかできない。これらの制約から、型 C に属するオブジェクトはすべて、クラス C の宣言（これは一意である）で定義されているインスタンス変数をすべて必ず持っていると保証される（さらに他のインスタンス変数を持っている可能性はある）。このようなオブジェクトのメソッドが他の C を引数として受け取り、引数のインスタンス変数に直接アクセスしても、そのインスタンス変数が C で定義されているものである限りは大丈夫である。これによって、和集合演算のような強二項演算をメソッドとして定義できるようになっている。この種の「ハイブリッド」なオブジェクトモデルは、Pierce and Turner [372] と Katiyar et al. [250] によって形式化された。Fisher and Mitchell [153; 154]; Fisher [151; 150] でより詳しく説明されている。

24.3 全称型で存在型を表現する

276 ページでは多相型として二つ組を表現したが、それと同じような方法で、全称型を使って存在型を表現できる。存在型の要素は型と値の二つ組であるという直観を用いる。

$$\{\exists X, T\} \stackrel{\text{def}}{=} \forall Y.\ (\forall X.\ T \rightarrow Y) \rightarrow Y$$

つまり存在型パッケージは、結果の型と**継続**とを与えると、最終的な結果を返すために継続を呼び出すデータ値だと考えられる。継続は、型 X と、T 型の値との二つを引数として受け取り、それらを使って最終的な結果を計算する。

このように存在型を表現すると決めると、パッケージ化とアンパッケージ化の構文の表現は必然的に決まってしまう。パッケージ化の構文

```
{*S,t} as {∃X,T}
```

を表現するには、S と t を使って ∀Y. (∀X. T→Y) → Y の値を作らなければならない。この型は全称量化子で始まり、その本体は矢印型である。よってこの型の要素は二つの抽象で始まるはずである。

$$\texttt{\{*S,t\} as \{∃X,T\}} \quad \stackrel{\text{def}}{=} \quad \texttt{λY. λf:(∀X.T→Y). ...}$$

最後に、型 Y の値を返す必要がある。そのためには、何か適当な引数に f を適用するしか方法がないのは明らかである。まず型 S を与える（今のところ他に渡せる型がないのだから、自然な選択である）。

$$\texttt{\{*S,t\} as \{∃X,T\}} \quad \stackrel{\text{def}}{=} \quad \texttt{λY. λf:(∀X.T→Y). f [S] ...}$$

ここで、型適用 f [S] は型 [X ↦ S](T→Y)、つまり型 ([X ↦ S]T)→Y を持つ。よって次の引数として t を与えられる（規則 T-Pack より、t は型 [X ↦ S]T を持つ）。

$$\texttt{\{*S,t\} as \{∃X,T\}} \quad \stackrel{\text{def}}{=} \quad \texttt{λY. λf:(∀X.T→Y). f [S] t}$$

この適用 f [S] t 全体としての型は、所要の通り Y となる。

アンパッケージ化の構文 let {X,x}=t_1 in t_2 も同様に表現できる。まず型付け規則 T-Unpack から、t_1 はある型 {∃X,T_{12}} を持っているはずだとわかる。また、t_2 は（束縛 X と x:T_{12} で拡張された文脈の下で）型 T_2 を持っているはずである。そして T_2 は、let...in... の式全体に期待している型である[†5]。ここでの直観は、273 ページの Church 表現と同じように、導入形式（{*S,t}）が「自分自身を除去する」という動作を表す値として表現されていることである。そのため、ここでの除去形式の表現は、存在型パッケージ t_1 をそのまま用いて、それを望む型 T_2 の結果になるまで引数に適用していくというものであるはずである。

$$\texttt{let \{X,x\}=}t_1\texttt{ in }t_2 \quad \stackrel{\text{def}}{=} \quad t_1 \texttt{ ...}$$

t_1 に与える第一引数は、式全体の返り値の型、つまり T_2 である。

$$\texttt{let \{X,x\}=}t_1\texttt{ in }t_2 \quad \stackrel{\text{def}}{=} \quad t_1 \texttt{ [}T_2\texttt{] ...}$$

適用 t_1 [T_2] は型 (∀X. T_{12}→T_2) → T_2 を持つ。つまり、あとは (∀X.T_{12}→T_2) 型の引数を与えられれば完了である。そのような引数は、本体 t_2 を変数 X と x に関して**抽象化する**ことで作れる。

$$\texttt{let \{X,x\}=}t_1\texttt{ in }t_2 \quad \stackrel{\text{def}}{=} \quad t_1 \texttt{ [}T_2\texttt{] (λX. λx:}T_{12}\texttt{. }t_2\texttt{)}$$

これで表現できた。

---

†5　このように項の構文に現れない型情報がこの変換で余分に必要になるのは、厳密に言うと、変換しようとしているものが実際には**型付け導出**であり、項ではないということを意味している。似たような状況は、15.6 節で、部分型付けに対する**型強制意味論**を定義するときにも見た。

演習 24.3.1. [推奨, ★★ ↛] 白紙を用意し、上の説明を見ないで、一から同じことをやってみよ。 □

演習 24.3.2. [★★★] この存在型の表現が正しいことを確かめるには、何を証明しないといけないだろうか。 □

演習 24.3.3. [★★★★] 逆に、存在型を使って全称型を表現することはできるだろうか。 □

24.4 注記

ADT と存在型の対応関係を最初に研究したのは Mitchell と Plotkin である [325]。彼らはオブジェクトとの関係にも言及した。Pierce と Turner はこの関係を精緻化した [373]。詳細とさらなる参考文献に関しては、第 32 章を見よ。オブジェクトと ADT のトレードオフは、Reynolds [409]、Cook [111]、Bruce et al. [61]、その他大勢によって議論されてきた。特に Bruce et al. [61] は、二項メソッドに関する長い議論である。

本章では、単純な抽象データ型のための自然な型理論的基盤として、存在型がどのように使えるかを説明してきた。ML のような言語に見られる**モジュールシステム**のように、本節の話と関係はするがずっと強力なものを説明するため、より洗練された機構が多数研究されてきた。この分野について知りたければ、Cardelli and Leroy [84]、Leroy [279]、Harper and Lillibridge [196]、Lillibridge [287]、Harper and Stone [199]、Crary et al. [122] から読み始めるとよい。

型構造は、抽象のレベルを守らせるための構文的な規律である。

—*John Reynolds, 1983* [413]

第 25 章

System F の ML 実装

本章では第 10 章の $\lambda_{\to}$ の実装を拡張し第 23 章と第 24 章の全称型と存在型を含める。この型システムを定義する規則は構文主導である（これは $\lambda_{\to}$ 自体と同様で、部分型付けや同値再帰型のある計算体系とは異なる）。そのため、OCaml による実現はかなり単純である。$\lambda_{\to}$ 実装に対する最も興味深い拡張は変数束縛を（量化子の位置に）含む型の表現である。これらについて、第 6 章で導入した **de Bruijn インデックス**の技法を用いる。

25.1 型の名無し表現

まず型の構文を型変数と全称量化子と存在量化子で拡張する。

```
type ty =
    TyVar of int * int
  | TyArr of ty * ty
  | TyAll of string * ty
  | TySome of string * ty
```

ここでは 7.1 節における項の表現と全く同様の慣習を用いている。型変数は二つの整数からなる。一つ目が変数の束縛子までの距離を指定する一方で、二つ目は一貫性検査のために期待される文脈のサイズの合計を指定している。量化子は束縛する変数の名前の文字列で注釈されており、関数を表示するときの手がかりとなる。

次に、binding 型に新しい構築子を追加することにより文脈を拡張し、項変数に加えて型変数の束縛を保持させるようにする。

```
type binding =
    NameBind
  | VarBind of ty
  | TyVarBind
```

これまでの実装と同様に、束縛子 NameBind は構文解析関数と表示関数でのみ用いられる。構築子 VarBind も以前と同様に、型を保持する。新しい構築子 TyVarBind は特にデータ値を持たないが、これは（項変数と違って）この型システムの型変数は特別な仮定で注釈されていないためである。有界量化（第 26 章）や高階のカインド（第 29 章）を持つ型システムでは、各 TyVarBind に適切な注釈を加えることになる。

25.2 型のシフトと代入

型が変数を含むようになったため、型のシフトと代入のための関数を定義する必要がある。

演習 25.2.1. [★]　定義 6.2.1（60 ページ）の、項をシフトする関数を手本として、型の中の変数をシフトする同様の関数の数学的な定義を書け。 □

7.2 節では、項のシフトと代入を別々の関数で示したが、本書のウェブサイトで得られる実装では、実際には両方の仕事を行う総称的な「マッピング」関数を使っていることも述べた。同様のマッ

ピング関数を使えば、型のシフトと代入を定義できる。ここで、これらのマッピング関数を見てみよう。

基本的な観察は、シフトの振る舞いと代入の振る舞いが、変数を除いたすべての構築子において全く同じことである。変数における振る舞いを抽象化すると、これらは全く同一になる。例えば、次は演習 25.2.1 の解答を OCaml に機械的に書き写すことで得られる、型のシフト専用の関数である。

```
let typeShiftAbove d c tyT =
  let rec walk c tyT = match tyT with
      TyVar(x,n) → if x>=c then TyVar(x+d,n+d) else TyVar(x,n+d)
    | TyArr(tyT1,tyT2) → TyArr(walk c tyT1,walk c tyT2)
    | TyAll(tyX,tyT2) → TyAll(tyX,walk (c+1) tyT2)
    | TySome(tyX,tyT2) → TySome(tyX,walk (c+1) tyT2)
  in walk c tyT
```

この関数の引数は、自由変数をシフトする量 d、それ未満ではシフトを**しない**打ち切り値 c（その型の中の量化子で束縛された変数のシフトを避けるため）、そして シフトされる型 tyT である。

ここで、typeShiftAbove から TyVar の節を新たな引数 onvar で抽象化し、TyVar の節でしか言及されない引数 d を削除すると、総称的なマッピング関数

```
let tymap onvar c tyT =
  let rec walk c tyT = match tyT with
    TyArr(tyT1,tyT2) → TyArr(walk c tyT1,walk c tyT2)
  | TyVar(x,n) → onvar c x n
  | TyAll(tyX,tyT2) → TyAll(tyX,walk (c+1) tyT2)
  | TySome(tyX,tyT2) → TySome(tyX,walk (c+1) tyT2)
  in walk c tyT
```

が得られる。これに TyVar 節を（c、x、n で抽象化された関数として）引数に与えればシフト関数が復元できる。

```
let typeShiftAbove d c tyT =
  tymap
    (fun c x n → if x>=c then TyVar(x+d,n+d) else TyVar(x,n+d))
    c tyT
```

はじめの打ち切り値が 0 の場合に特化した typeShiftAbove を定義しておくのも便利である。

```
let typeShift d tyT = typeShiftAbove d 0 tyT
```

tymap を具体化すれば、型 tyT 中の数 j が振られた型変数に型 tyS を代入する操作も実装できる。

```
let typeSubst tyS j tyT =
  tymap
    (fun j x n → if x=j then (typeShift j tyS) else (TyVar(x,n)))
    j tyT
```

型検査や評価の間に型代入を用いるときは、常に 0 番目の（最も外側の）変数に代入し、その変数が消えるように結果をシフトしたくなる。補助関数 typeSubstTop がこれを行ってくれる。

```
let typeSubstTop tyS tyT =
  typeShift (-1) (typeSubst (typeShift 1 tyS) 0 tyT)
```

25.3 項

項のレベルで必要な拡張も同様である。まず第 10 章の term データ型を、全称型と存在型の導入形式および除去形式で拡張する。

```
type term =
    TmVar of info * int * int
  | TmAbs of info * string * ty * term
  | TmApp of info * term * term
  | TmTAbs of info * string * term
  | TmTApp of info * term * ty
  | TmPack of info * ty * term * ty
  | TmUnpack of info * string * string * term * term
```

項のシフトと代入の定義は、第 10 章での定義と同様である。しかしながら、ここでは前節で型に対して行ったように、共通の総称的なマッピング関数の方法で書いてみよう。マッピング関数は次のようになる。

```
let tmmap onvar ontype c t =
  let rec walk c t = match t with
    TmVar(fi,x,n) → onvar fi c x n
  | TmAbs(fi,x,tyT1,t2) → TmAbs(fi,x,ontype c tyT1,walk (c+1) t2)
  | TmApp(fi,t1,t2) → TmApp(fi,walk c t1,walk c t2)
  | TmTAbs(fi,tyX,t2) → TmTAbs(fi,tyX,walk (c+1) t2)
  | TmTApp(fi,t1,tyT2) → TmTApp(fi,walk c t1,ontype c tyT2)
  | TmPack(fi,tyT1,t2,tyT3) →
      TmPack(fi,ontype c tyT1,walk c t2,ontype c tyT3)
  | TmUnpack(fi,tyX,x,t1,t2) →
      TmUnpack(fi,tyX,x,walk c t1,walk (c+2) t2)
  in walk c t
```

tmmap は、tymap より一つ多い、**四つ**の引数を取ることに注意されたい。この理由は、項が二つの異なる種類の変数を含みうることに気づけばわかるはずである。項変数と、項の型注釈に埋め込まれた型変数である。つまり、例えばシフトの間に実際に処理すべきなのは項変数と型変数という二種類の「葉」なのである。ontype 引数は、TmAbs の場合のように型注釈を含む項構築子を処理するときに何をすべきかを項のマッピング関数に教える。もっと大きい言語を扱っていたとしたら、同様の処理を行う場合がさらに幾つかあっただろう。

項のシフトは tmmap に適切な引数を与えることにより定義できる。

```
let termShiftAbove d c t =
  tmmap
    (fun fi c x n → if x>=c then TmVar(fi,x+d,n+d)
                    else TmVar(fi,x,n+d))
    (typeShiftAbove d)
    c t

let termShift d t = termShiftAbove d 0 t
```

項変数については、typeShiftAbove でそうしたように、打ち切り値を調べて新しい変数を作る。型については、前節で定義した型のシフト関数を呼ぶ。

ある項に別の項を代入する関数も同様である。

```
let termSubst j s t =
  tmmap
    (fun fi j x n → if x=j then termShift j s else TmVar(fi,x,n))
    (fun j tyT → tyT)
    j t
```

termSubst は型注釈を変えないことに注意されたい（型は項変数を含むことはないため、項の代入は型に影響しない）。

型を項に代入する関数も必要である。これは、例えば型適用のための評価規則で用いられる。

$$(\lambda X.t_{12})\ [T_2] \longrightarrow [X \mapsto T_2]t_{12} \qquad \text{(E-TAPPTABS)}$$

この関数もまた項のマッピング関数を用いて定義できる。

```
let rec tytermSubst tyS j t =
  tmmap (fun fi c x n → TmVar(fi,x,n))
        (fun j tyT → typeSubst tyS j tyT) j t
```

今回は、項変数を扱うために tmmap に渡す関数は恒等関数である（元の項変数を単に再構築する）。型注釈に辿り着いたとき、その型に型レベルの代入を行う。

最後に、型でそうしたのと同様に、eval と typeof 関数で使う基本的な代入関数をパッケージ化した補助関数を定義する。

```
let termSubstTop s t =
  termShift (-1) (termSubst 0 (termShift 1 s) t)
let tytermSubstTop tyS t =
  termShift (-1) (tytermSubst (typeShift 1 tyS) 0 t)
```

25.4 評価

eval 関数の拡張は図 23-1 と図 24-1 で導入した評価規則の単純な写しである。困難な仕事は前節で定義した代入関数が行ってくれる。

```
let rec eval1 ctx t = match t with
    ...
  | TmTApp(fi,TmTAbs(_,x,t11),tyT2) →
      tytermSubstTop tyT2 t11
  | TmTApp(fi,t1,tyT2) →
      let t1' = eval1 ctx t1 in
      TmTApp(fi, t1', tyT2)
  | TmUnpack(fi,_,_,TmPack(_,tyT11,v12,_),t2) when isval ctx v12 →
      tytermSubstTop tyT11 (termSubstTop (termShift 1 v12) t2)
  | TmUnpack(fi,tyX,x,t1,t2) →
      let t1' = eval1 ctx t1 in
      TmUnpack(fi,tyX,x,t1',t2)
  | TmPack(fi,tyT1,t2,tyT3) →
      let t2' = eval1 ctx t2 in
      TmPack(fi,tyT1,t2',tyT3)
    ...
```

演習 25.4.1. [★]　最初の TmUnpack の場合に termShift が必要なのはなぜか。　□

25.5 型付け

typeof 関数の新しい節もまた、型抽象と型適用および存在型のパッケージ化とアンパッケージ化に対する型付け規則から直接的に得られる。ここで、新しい TmTAbs と TmTApp の節を以前の普通の関数抽象と関数適用の節と比較できるよう、typeof の全定義を示す。

```
let rec typeof ctx t =
  match t with
    TmVar(fi,i,_) → getTypeFromContext fi ctx i
  | TmAbs(fi,x,tyT1,t2) →
      let ctx' = addbinding ctx x (VarBind(tyT1)) in
      let tyT2 = typeof ctx' t2 in
      TyArr(tyT1, typeShift (-1) tyT2)
  | TmApp(fi,t1,t2) →
      let tyT1 = typeof ctx t1 in
      let tyT2 = typeof ctx t2 in
      (match tyT1 with
          TyArr(tyT11,tyT12) →
            if (=) tyT2 tyT11 then tyT12
            else error fi "parameter type mismatch"
        | _ → error fi "arrow type expected")
  | TmTAbs(fi,tyX,t2) →
      let ctx = addbinding ctx tyX TyVarBind in
      let tyT2 = typeof ctx t2 in
      TyAll(tyX,tyT2)
  | TmTApp(fi,t1,tyT2) →
      let tyT1 = typeof ctx t1 in
      (match tyT1 with
           TyAll(_,tyT12) → typeSubstTop tyT2 tyT12
         | _ → error fi "universal type expected")
  | TmPack(fi,tyT1,t2,tyT) →
      (match tyT with
          TySome(tyY,tyT2) →
            let tyU = typeof ctx t2 in
            let tyU' = typeSubstTop tyT1 tyT2 in
            if (=) tyU tyU' then tyT
            else error fi "doesn't match declared type"
        | _ → error fi "existential type expected")
  | TmUnpack(fi,tyX,x,t1,t2) →
      let tyT1 = typeof ctx t1 in
      (match tyT1 with
          TySome(tyY,tyT11) →
            let ctx' = addbinding ctx tyX TyVarBind in
            let ctx" = addbinding ctx' x (VarBind tyT11) in
            let tyT2 = typeof ctx" t2 in
            typeShift (-2) tyT2
        | _ → error fi "existential type expected")
```

新しい節の中で最も興味深いのは、TmUnpack の節である。これは次のステップを辿っている。(1) 部分式 t_1 を検査し存在型 $\{\exists X.T_{11}\}$ を持つことを確認する。(2) 文脈 Γ を型変数の束縛 X と項変数の束縛 $x{:}T_{11}$ で拡張し、t_2 が何らかの型 T_2 を持つことを検査する。(3) T_2 中の自由変数のインデックスを下に二つシフトし、元の Γ の下で意味をなすようにする。(4) 結果の型を let...in...

式全体の型として返す。

X が T_2 に自由に出現するならば、ステップ (3) のシフトが、負のインデックスを持つ自由変数を含んだ無意味な型を生成するのは明らかである。型検査はこの時点で失敗しなくてはならない。このことは、**typeShiftAbove** を再定義し、負のインデックスを持つ型変数を構築しようとしたことを検出し、無意味な値を返す代わりにエラーを出すようにすれば保証できる。

```
let typeShiftAbove d c tyT =
  tymap
    (fun c x n → if x>=c then
                   if x+d<0 then err "Scoping error!"
                   else TyVar(x+d,n+d)
                else TyVar(x,n+d))
    c tyT
```

この検査により、存在型の除去形式 `let {X,x}=`t_1 `in` t_2 の本体 t_2 について、求めようとする型が束縛された型変数 X を含むときはいつでも、スコープの扱いについてエラーを報告するようになる。

```
  let {X,x}=({*Nat,0} as {∃X,X}) in x;
▶ Error: Scoping error!
```

第 26 章

有界量化

本章のほとんどで扱う体系は純粋な $F_{<:}$（図 26-1）である。例ではさらにレコード（図 11-7）と数（図 8-2）を使う。対応する OCaml 実装は fullfsub と fullfomsub である。（ほとんどの例は fullfsub 型検査器で十分であり、fullfomsub はパラメータを持つ型の省略（例えば Pair）に関する型検査で必要になる。）

プログラミング言語における興味深い課題の多くは、一つ一つでは比較的単純な機能が相互作用するときに生じる。本章で導入する**有界量化**は、多相性と部分型付けを組み合わせるときに生じ、型システムの表現力とメタ理論的な複雑さが本質的に増加する。ここで調べる体系は $F_{<:}$（「エフ・サブ」）と呼ばれ、1980 年代中盤に提案されて以来、プログラミング言語研究、特にオブジェクト指向プログラミングの基礎研究において中心的な役割を果たしてきた。

26.1 動機

部分型付けと多相性を組み合わせる場合、完全に直交する機能として扱うのが最も単純である。つまり、第 15 章と第 23 章の型システムの実質的な和集合を考えるのである。この型システムは理論的に問題なく、さらに部分型付けと多相性が個別に有用であるのと同じ理由で、有用である。しかし、ひとたび両方の機能が同じ言語に導入されると、より面白い方法で混ぜたくなる。それを示すために、とても簡単な例を考えよう。（のちに 26.3 節で別の例を、第 27 章と第 32 章ではより大きい実践的な事例を扱う。）

f を、数値フィールド a を持つレコードの恒等関数とする。

```
  f = λx:{a:Nat}. x;
▸ f : {a:Nat} → {a:Nat}
```

ra がフィールド a を持つレコード

```
  ra = {a=0};
```

ならば、以前の章で扱ったどんな型システムにおいても、ra に f を適用でき、同じ型のレコードが返される。

```
  f ra;
▸ {a=0} : {a:Nat}
```

同様に、二つのフィールド a と b を持つ、もう少し大きなレコード rab を定義すると、

```
  rab = {a=0, b=true};
```

包摂の規則（図 15-1 の T-Sub）を使えば、rab の型を f が期待する型 {a:Nat} に昇格でき、f を rab に適用できる。

```
  f rab;
▸ {a=0, b=true} : {a:Nat}
```

しかしながら、適用の結果の型はフィールド a しか持たず、(f rab).b のような項は型付け不能と判断される。言い換えると、恒等関数に rab を渡すとそのフィールド b にアクセスする能力が失われてしまう。

System F の多相性を用いると、f を違ったやり方で書ける。

```
  fpoly = λX. λx:X. x;
▶ fpoly : ∀X. X → X
```

fpoly を rab（および適切な型引数）に適用すれば、望みの結果が得られる。

```
  fpoly [{a:Nat, b:Bool}] rab;
▶ {a=0, b=true} : {a:Nat, b:Bool}
```

しかしながらこのやり方では、x の型を変数にするとき、使いたかったかもしれない情報を捨ててしまっている。例えば、元の引数と、そのフィールド a の後者値との二つ組を返すバージョンの f を書きたいとしよう。

```
  f2 = λx:{a:Nat}. {orig=x, asucc=succ(x.a)};
▶ f2 : {a:Nat} → {orig:{a:Nat}, asucc:Nat}
```

ここでまた、部分型付けを使えば、f2 を ra と rab の両方に適用できるが、後者の場合はフィールド b が失われてしまう。

```
  f2 ra;
▶ {orig={a=0}, asucc=1} : {orig:{a:Nat}, asucc:Nat}

  f2 rab;
▶ {orig={a=0,b=true}, asucc=1} : {orig:{a:Nat}, asucc:Nat}
```

しかし今回は多相性を使っても解決にならない。前と同じように x の型を変数 X で置き換えると、x はフィールド a を持つレコードであるという制約が失われてしまう。しかしその制約は、返り値の asucc フィールドを計算するために必要なのである。

```
  f2poly = λX. λx:X. {orig=x, asucc=succ(x.a)};
▶ Error: Expected record type
```

型で表現したい f2 の操作的振る舞いとは、それが数値のフィールド a を含む任意のレコード型 R の引数を取り、型 R のフィールドと型 Nat のフィールドを持つレコードを結果として返すことである。これは部分型関係を使えば簡潔に表現できる。f2 は型 {a:Nat} の任意の部分型 R を持つ引数を取り、型 R のフィールドと型 Nat のフィールドを持つレコードを返す。この直観は f2poly の束縛変数 X に**部分型付け制約**を導入することで形式化できる。

```
  f2poly = λX<:{a:Nat}. λx:X. {orig=x, asucc=succ(x.a)};
▶ f2poly : ∀X<:{a:Nat}. X → {orig:X, asucc:Nat}
```

このいわゆる**有界量化**が System $F_{<:}$ の特徴的な機能である。

26.2 定義

形式的には、$F_{<:}$ は第 23 章の System F の型および項と第 15 章の部分型関係を組み合わせ、さらに全称量化子が部分型付け制約を保持するように詳細化することで得られる。有界な存在量化子も同様に定義でき、26.5 節で扱う。

$F_{<:}$ の部分型関係を定義する理にかなった方法は実際のところ二つあり、有界量化子を比較する規則（S-ALL）の定式化が異なる。より扱いやすいが柔軟性に欠ける**カーネル**規則と、より表現力があるが技術的にいくぶん問題をはらむ**フル**部分型付け規則である。以降の小節で両方を詳細に議論する。最初の幾つかの小節でカーネルを、309 ページでフルを導入する。どちらについて述べているのかを正確にする必要があるときは、これらの規則を持つ体系全体をそれぞれ**カーネル** $F_{<:}$ と**フル** $F_{<:}$ と呼ぶ。限定のない $F_{<:}$ は両方の体系を指す。

図 26-1 にカーネル $F_{<:}$ の全定義を示す。以前の体系との違いには網掛けを施してある。

有界量化と非有界量化

図より直ちに明らかな点は $F_{<:}$ の構文が有界量化**のみ**を提供していることであり、純粋な System F の、通常の非有界な量化は消え失せている。この理由はそれが必要ないからである。上界が Top である有界量化子は Top のすべての部分型の上、つまりすべての型の上を動く。それゆえ非有界な量化は次の略記で得られる。

$$\forall\texttt{X.T} \quad \stackrel{\text{def}}{=} \quad \forall\texttt{X<:Top.T}$$

この略記は以降で頻繁に用いる。

スコープの扱い

図 26-1 では明らかではない重要な技術的詳細として、型変数のスコープに関する点がある。明らかに、$\Gamma \vdash$ **t : T** の形の型付け判断式について考えるときはいつでも、t と T の自由変数が Γ の定義域に入っていることを意図している。しかし、Γ の**中に**現れる自由な型変数についてはどうだろうか? 特に、次の文脈のうちどれが正しいスコープを持つと考えるべきだろうか?

$$\begin{array}{lll}\Gamma_1 & = & \texttt{X<:Top, y:X}\to\texttt{Nat}\\ \Gamma_2 & = & \texttt{y:X}\to\texttt{Nat, X<:Top}\\ \Gamma_3 & = & \texttt{X<:\{a:Nat,b:X\}}\\ \Gamma_4 & = & \texttt{X<:\{a:Nat,b:Y\}, Y<:\{c:Bool,d:X\}}\end{array}$$

Γ_1 は疑いなく正しいスコープを持つ。まず型変数 X を、そして型に X を伴う項変数 y を導入している。型検査の最中にこの文脈を生むかもしれない項は λX<:Top. λy:X→Nat. t の形を持つ。y の型の中の X が外側の λ で束縛されているのは明らかである。他方、同様の推論では Γ_2 は誤っているように見える。なぜならこの文脈を生む項、例えば λy:X→Nat. λX<:Top. t といった項では、X に期待されるスコープが何であるか明らかでないためである。

より興味深いケースは Γ_3 である。λX<:{a:Nat,b:X}. t のような項においては、二番目の X が束縛されているため明白**である**、と論じることもできる。ここで X の束縛のスコープがそれ自身の上界(と、いつもの通り、束縛の右にあるすべてのもの)を含むとみなせばよい。この改良を伴う有界量化は **F 有界量化**(Canning, Cook, Hill, Olthoff, and Mitchell [71])と呼ばれる。F 有界量化はオブジェクト指向プログラミングのための型の議論によく現れ、GJ の言語設計(Bracha, Odersky, Stoutamire, and Wadler [54])で用いられた。しかしながら、その理論は通常の $F_{<:}$ (Ghelli [174]; Baldan, Ghelli, and Raffaetà [36])よりいくぶん複雑であり、体系に再帰型が含まれる場合にのみ本当に面白くなる(再帰しない型 X は X<:{a:Nat,b:X} のような制約を満たすことはない)。

Γ_4 のようなより一般的な文脈では、上界を介して型変数同士の相互再帰を認めているが、全くない話ではない。そのような計算では、新しい変数束縛がその変数と既存の変数を含む任意の不等式を一般に導入しうる。

F 有界量化について以降では考慮せず、$\Gamma_2, \Gamma_3, \Gamma_4$ はすべて正しいスコープを持たないとする。より形式的には、ある文脈における型 T に言及するときはいつでも、T の自由変数が T の出現より左の文脈の部分で束縛されていなければならないとする。

→ ∀ <: Top　　System F（図 23-1）と単純型付け（図 15-1）に基づく

構文

t ::=		項：
	x	変数
	λx:T.t	ラムダ抽象
	t t	関数適用
	λX<:T.t	型抽象
	t [T]	型適用
v ::=		値：
	λx:T.t	ラムダ抽象値
	λX<:T.t	型抽象値
T ::=		型：
	X	型変数
	Top	最大の型
	T→T	関数の型
	∀X<:T.T	全称型
Γ ::=		文脈：
	∅	空の文脈
	Γ, x:T	項変数の束縛
	Γ, X<:T	型変数の束縛

評価　$t \longrightarrow t'$

$$\frac{t_1 \longrightarrow t_1'}{t_1\ t_2 \longrightarrow t_1'\ t_2} \quad \text{(E-APP1)}$$

$$\frac{t_2 \longrightarrow t_2'}{v_1\ t_2 \longrightarrow v_1\ t_2'} \quad \text{(E-APP2)}$$

$$\frac{t_1 \longrightarrow t_1'}{t_1\ [T_2] \longrightarrow t_1'\ [T_2]} \quad \text{(E-TAPP)}$$

$$(\lambda X{<:}T_{11}.t_{12})\ [T_2] \longrightarrow [X \mapsto T_2]t_{12} \quad \text{(E-TAPPTABS)}$$

$$(\lambda x{:}T_{11}.t_{12})\ v_2 \longrightarrow [x \mapsto v_2]t_{12} \quad \text{(E-APPABS)}$$

部分型付け　$\Gamma \vdash S <: T$

$$\Gamma \vdash S <: S \quad \text{(S-REFL)}$$

$$\frac{\Gamma \vdash S <: U \qquad \Gamma \vdash U <: T}{\Gamma \vdash S <: T} \quad \text{(S-TRANS)}$$

$$\Gamma \vdash S <: Top \quad \text{(S-TOP)}$$

$$\frac{X{<:}T \in \Gamma}{\Gamma \vdash X <: T} \quad \text{(S-TVAR)}$$

$$\frac{\Gamma \vdash T_1 <: S_1 \qquad \Gamma \vdash S_2 <: T_2}{\Gamma \vdash S_1 \to S_2 <: T_1 \to T_2} \quad \text{(S-ARROW)}$$

$$\frac{\Gamma, X{<:}U_1 \vdash S_2 <: T_2}{\Gamma \vdash \forall X{<:}U_1.S_2 <: \forall X{<:}U_1.T_2} \quad \text{(S-ALL)}$$

型付け　$\Gamma \vdash t : T$

$$\frac{x{:}T \in \Gamma}{\Gamma \vdash x : T} \quad \text{(T-VAR)}$$

$$\frac{\Gamma, x{:}T_1 \vdash t_2 : T_2}{\Gamma \vdash \lambda x{:}T_1.t_2 : T_1 \to T_2} \quad \text{(T-ABS)}$$

$$\frac{\Gamma \vdash t_1 : T_{11} \to T_{12} \qquad \Gamma \vdash t_2 : T_{11}}{\Gamma \vdash t_1\ t_2 : T_{12}} \quad \text{(T-APP)}$$

$$\frac{\Gamma, X{<:}T_1 \vdash t_2 : T_2}{\Gamma \vdash \lambda X{<:}T_1.t_2 : \forall X{<:}T_1.T_2} \quad \text{(T-TABS)}$$

$$\frac{\Gamma \vdash t_1 : \forall X{<:}T_{11}.T_{12} \qquad \Gamma \vdash T_2 <: T_{11}}{\Gamma \vdash t_1\ [T_2] : [X \mapsto T_2]T_{12}} \quad \text{(T-TAPP)}$$

$$\frac{\Gamma \vdash t : S \qquad \Gamma \vdash S <: T}{\Gamma \vdash t : T} \quad \text{(T-SUB)}$$

▲ 図 26-1.　有界量化（カーネル $F_{<:}$）

部分型付け

$F_{<:}$ の型変数は（ちょうど通常の項が型を伴うように）上界を伴う。これらの上界は部分型付けと型検査の両方で把握しておかなくてはならない。このため、各型変数に対して上界を含めるように文脈における型の束縛を変更する。これらの上界は部分型付けの最中に「仮定より型変数 X は型 T の部分型である」という形のステップを正当化するために用いる。

$$\frac{X<:T \in \Gamma}{\Gamma \vdash X <: T} \qquad \text{(S-TVAR)}$$

この規則を追加することは、部分型付けが三**項関係**になることを意味する。すなわち、すべての部分型付け判断式は $\Gamma \vdash S <: T$ の形となる。これは、「Γ の仮定の下で S は T の部分型である」と読む。残りすべての部分型付け規則に文脈を加えることでこの改良を仕上げる（図 26-1 参照)。

変数に関する新しい規則だけでなく、量化された型を比較する部分型付け規則（S-ALL）も加えなければならない。図 26-1 はより単純な**カーネル**規則と呼ばれる変種を与えている。この規則では、比較される二つの量化子の上界が同一でなければならない。

$$\frac{\Gamma, X<:U_1 \vdash S_2 <: T_2}{\Gamma \vdash \forall X<:U_1.S_2 <: \forall X<:U_1.T_2} \qquad \text{(S-ALL)}$$

「カーネル」という用語は Cardelli と Wegner の原論文 [88] に由来し、そこではこの $F_{<:}$ の変種は **Kernel Fun** と呼ばれている

演習 26.2.1. [★ ↛] 文脈 Γ = B<:Top, X<:B, Y<:X のもとで B→Y <: X→B を示す部分型付け導出木を書け。 □

型付け

通常の全称型の型付け規則も改良しなければならない。これらの拡張は単純である。有界量化子の導入規則では本体の型検査中に上界をラムダ抽象から文脈に移す。

$$\frac{\Gamma, X<:T \vdash t_2 : T_2}{\Gamma \vdash \lambda X<:T.t_2 : \forall X<:T.T_2} \qquad \text{(T-TABS)}$$

除去規則では渡された型引数が実際に上界を満たすことを検査する。

$$\frac{\Gamma \vdash t_1 : \forall X<:T_{11}.T_{12} \qquad \Gamma \vdash T_2 <: T_{11}}{\Gamma \vdash t_1\ [T_2] : [X \mapsto T_2]T_{12}} \qquad \text{(T-TAPP)}$$

フル $F_{<:}$

カーネル $F_{<:}$ においては、二つの量化された型は上界が同一であるときのみ比較できる。もし量化子を矢印型の一種（その要素が型から項への関数であるもの）であるとみなすならば、カーネル規則は矢印型のための標準的な部分型付け規則の「共変」制限に対応する。共変制限とは、矢印型の定義域が部分型において変化することを認めないという制限である。

$$\frac{S_2 <: T_2}{U \to S_2 <: U \to T_2}$$

→ ∀ <: Top フル　　　　$F_{<:}$（図 26-1）の拡張

新しい部分型付け規則　$\boxed{\Gamma \vdash S <: T}$

$$\frac{\Gamma \vdash T_1 <: S_1 \qquad \Gamma, X<:T_1 \vdash S_2 <: T_2}{\Gamma \vdash \forall X<:S_1.S_2 <: \forall X<:T_1.T_2} \quad \text{(S-ALL)}$$

▲ 図 26-2. 「フル」の有界量化

この制限は矢印型と量化子の両方にとってかなり不自然に見える。この類推は、図 26-2 に示す通り有界量化子の「左側」における反変の部分型付けを認めるようカーネルの S-ALL 規則を改良すべきことを示唆している。

直観的には、フル版の S-ALL は次のように理解できる。型 $T = \forall X<:T_1.T_2$ は型から値への関数の集まりを表しており、それぞれが T_1 の部分型を T_2 の具体例に写す。もし T_1 が S_1 の部分型ならば、T の定義域は $S = \forall X<:S_1.S_2$ の定義域より小さい。それゆえ S はより強い制約であり多相的な値のより小さな集まりを表している。さらに、もし、両方の集まりに含まれる関数が受け取れる型 U（つまり、より厳格な要求 $U <: T_1$ を満たすもの）について、S を U で具体化した S_2 が、T を U で具体化した T_2 の部分型ならば、S は「各引数において強い」制約であり、やはり値のより小さい集まりを表している。

量化された型に関するカーネルの部分型付け規則しか持たない型システムはカーネル $F_{<:}$ と呼ばれる。フルの量化子の部分型付け規則を持つ型システムはフル $F_{<:}$ と呼ばれる。単に $F_{<:}$ という名前は両方の型システムを曖昧に指している。

演習 26.2.2. [★ ↛]　フル $F_{<:}$ の部分型関係で関連付けられるが、カーネル $F_{<:}$ ではそうならない型の二つ組の例を幾つか挙げよ。 □

演習 26.2.3. [★★★★]　この性質を必要とする**便利な**例を何か見つけられるか。 □

26.3 例

本節では $F_{<:}$ におけるプログラミングの小さな例を幾つか示す。これらの例は、実践的な応用を説明するというより、型システムの性質を例示することを意図している。より大きく洗練された例は以降の章（第 27 章と第 32 章）にある。本章のすべての例はカーネルとフル $F_{<:}$ の両方で有効である。

直積の表現

276 ページでは、数の二つ組の System F における表現を与えた。この表現は任意の型の二つ組に容易に一般化できる。型

```
Pair T₁ T₂ = ∀X. (T₁→T₂→X) → X;
```

の要素は T_1 と T_2 の二つ組を表している。構築子 `pair` と分解子 `fst` と `snd` は次のように定義される。(`pair` の定義における型指定は型検査器が読みやすい形で型を表示するのに役立っている。)

```
  pair = λX. λY. λx:X. λy:Y. (λR. λp:X→Y→R. p x y) as Pair X Y;
▸ pair : ∀X. ∀Y. X → Y → Pair X Y

  fst = λX. λY. λp: Pair X Y. p [X] (λx:X. λy:Y. x);
▸ fst : ∀X. ∀Y. Pair X Y → X

  snd = λX. λY. λp: Pair X Y. p [Y] (λx:X. λy:Y. y);
▸ snd : ∀X. ∀Y. Pair X Y → Y
```

$F_{<:}$ は System F のすべての機能を含むため、同じ表現は明らかに $F_{<:}$ でも使える。しかしより興味深いのは、この表現が幾つかの自然な部分型付けの性質をも持つことである。実際、二つ組に対して期待される部分型付け規則

$$\frac{\Gamma \vdash \mathsf{S}_1 <: \mathsf{T}_1 \qquad \Gamma \vdash \mathsf{S}_2 <: \mathsf{T}_2}{\Gamma \vdash \mathsf{Pair}\ \mathsf{S}_1\ \mathsf{S}_2 <: \mathsf{Pair}\ \mathsf{T}_1\ \mathsf{T}_2}$$

が、この表現から直ちに得られる。

演習 26.3.1. [★ ↛] このことを示せ。 □

レコードの表現

レコードとレコード型が、部分型付け規則も含めて純粋な $F_{<:}$ で実際に表現できるのを見るのは興味深い。ここで示す表現方法は Cardelli [80] によって発見された。

まずは**柔軟な組**を次のように定義する。これらは、通常の組と違い、部分型付け時に右へ拡張できるという意味で「柔軟」なのである。

定義 26.3.2. 各 $n \geq 0$ と型 T_1 から T_n について、

$$\{\mathsf{T}_i\ ^{i\in 1..n}\} \stackrel{\text{def}}{=} \mathsf{Pair}\ \mathsf{T}_1\ (\mathsf{Pair}\ \mathsf{T}_2\ \ldots\ (\mathsf{Pair}\ \mathsf{T}_n\ \mathsf{Top})\ldots)$$

とする。特に、$\{\} \stackrel{\text{def}}{=} \mathsf{Top}$ である。同様に、項 t_1 から t_n について、

$$\{\mathsf{t}_i\ ^{i\in 1..n}\} \stackrel{\text{def}}{=} \mathsf{pair}\ \mathsf{t}_1\ (\mathsf{pair}\ \mathsf{t}_2\ \ldots\ (\mathsf{pair}\ \mathsf{t}_n\ \mathsf{top})\ldots)$$

とする。ここで簡単のため pair への型引数は省略している。(ここで項 top は Top のどれか任意の要素、つまり任意の閉じた正しく型付けされた項である。) 射影 t.n は次のようになる (今回も型引数は省略する)。

$$\mathsf{fst}\ (\underbrace{\mathsf{snd}\ (\mathsf{snd}\ldots\ (\mathsf{snd}}_{n-1\ 個}\ \mathsf{t})\ldots)$$ □

この略記により、柔軟な組に対する部分型付けと型付けに関する次の規則が直ちに得られる。

$$\frac{\Gamma \vdash\ ^{i\in 1..n}\mathsf{S}_i <: \mathsf{T}_i}{\Gamma \vdash \{\mathsf{S}_i\ ^{i\in 1..n+k}\} <: \{\mathsf{T}_i\ ^{i\in 1..n}\}}$$

$$\frac{\Gamma \vdash\ ^{i\in 1..n}\mathsf{t}_i : \mathsf{T}_i}{\Gamma \vdash \{\mathsf{t}_i\ ^{i\in 1..n}\} : \{\mathsf{T}_i\ ^{i\in 1..n}\}}$$

$$\frac{\Gamma \vdash \mathsf{t} : \{\mathsf{T}_i\ ^{i\in 1..n}\}}{\Gamma \vdash \mathsf{t.i} : \mathsf{T}_i}$$

ここで、$\mathcal{L}$ をラベルの可算集合とし、全単射な関数 *label-with-index* : $\mathbb{N} \to \mathcal{L}$ で与えられる固定の全順序を持つとする。レコードを次のように定義する。

定義 26.3.3. L を $\mathcal{L}$ の有限部分集合とし、各 $l \in L$ について S_l を型とする。m を L の任意の要素のうち最大のインデックスとし、

$$\hat{\mathsf{S}}_i = \begin{cases} \mathsf{S}_l & \textit{label-with-index}(i) = l \in L \text{ の場合} \\ \mathsf{Top} & \textit{label-with-index}(i) \notin L \text{ の場合} \end{cases}$$

とする。レコード型 $\{l{:}\mathsf{S}_l{}^{l \in L}\}$ は柔軟な組 $\{\hat{\mathsf{S}}_i{}^{\,i \in 1..m}\}$ として定義される。同様に、各 $l \in L$ について t_l を項として、

$$\hat{\mathsf{t}}_i = \begin{cases} \mathsf{t}_l & \textit{label-with-index}(i) = l \in L \text{ の場合} \\ \mathsf{top} & \textit{label-with-index}(i) \notin L \text{ の場合} \end{cases}$$

とする。レコード値 $\{l{=}\mathsf{t}_l{}^{l \in L}\}$ は $\{\hat{\mathsf{t}}_i{}^{\,i \in 1..m}\}$ である。*label-with-index*$(i) = l$ のとき射影 `t.l` は組の射影 `t.i` である。 □

この表現はレコードの型付けと部分型付けに期待される規則（図 15-2 と図 15-3 の規則 S-RcdWidth、S-RcdDepth、S-RcdPerm、T-Rcd、T-Proj）を正当化する。しかしながら、興味は主に理論的なところにある。実践的な見地でいえばフィールドラベルの大域的な順序付けに依存することは深刻な欠点である。これは、分割コンパイルを伴う言語において、各モジュールで独立に数をラベルに割り当てることができず、すべてを同時に、つまりリンク時に割り当てなければならないことを意味する。

部分型付けを伴う Church 表現

$F_{<:}$ の表現能力に関する最後の例として、System F における Church 数の表現（273 ページ）に有界量化を加えると何が起こるか見てみよう。Church 数の型は次の通りだった。

```
CNat = ∀X. (X→X) → X → X;
```

この型の直観的で擬人的な読み方は、「結果の型 T を言いなさい。そこで T の上の関数と T の『基本要素』をくれたなら、その関数を基本要素に対して *n* 回反復して得られた別の T の要素を返しましょう」である。

これは二つの有界量化子を加え引数 s と z の型を詳細化することにより一般化できる。

```
SNat = ∀X<:Top. ∀S<:X. ∀Z<:X. (X→S) → Z → X;
```

直観的には、この型は次のように読む。「総称的な結果の型 X とその部分型 S と Z を私に与えなさい。そして集合 X 全体から部分集合 S へ写す関数と、特別な集合 Z の要素をくれたら、その関数を基本要素に対して *n* 回反復して得られた別の X の要素を返しましょう。」

なぜこれが面白いのかを見るために、わずかに異なる次の型について考える。

```
SZero = ∀X<:Top. ∀S<:X. ∀Z<:X. (X→S) → Z → Z;
```

SZero は SNat とほとんど同じ形であるものの、その要素の振る舞いについて何かもっと強いことをいっている。というのは、最終的な結果の型が単なる X の要素ではなく Z の要素であると約束しているからである。実際のところ、それが起こる方法はたった一つ、つまり z それ自身を返すことである。言い換えると、値

```
  szero = λX. λS<:X. λZ<:X. λs:X→S. λz:Z. z;
▸ szero : SZero
```

は型 SZero における**ただ一つの**値なのである（他のすべての SZero の要素が szero と同じに振る舞うという意味において）。SZero は SNat の部分型であるため、**szero : SNat** でもある。

一方、同様の型

```
  SPos = ∀X<:Top. ∀S<:X. ∀Z<:X. (X→S) → Z → S;
```

はより多くの値を持つ。例えば、

```
  sone   = λX. λS<:X. λZ<:X. λs:X→S. λz:Z. s z;
  stwo   = λX. λS<:X. λZ<:X. λs:X→S. λz:Z. s (s z);
  sthree = λX. λS<:X. λZ<:X. λs:X→S. λz:Z. s (s (s z));
```

などである。実際、SPos は szero **を除いて** SNat のすべての要素を値に持つ。

Church 数上の操作の型付けも同様に詳細化できる。例えば、型システムを使って後者関数が常に正の数を返すことを検査できる。

```
  ssucc = λn:SNat.
              λX. λS<:X. λZ<:X. λs:X→S. λz:Z.
                s (n [X] [S] [Z] s z);
▸ ssucc : SNat → SPos
```

同様に、引数の型を詳細化することで、関数 plus について型検査器が型 SPos→SPos→SPos を付けるように書ける。

```
  spluspp = λn:SPos. λm:SPos.
                λX. λS<:X. λZ<:X. λs:X→S. λz:Z.
                  n [X] [S] [S] s (m [X] [S] [Z] s z);
▸ spluspp : SPos → SPos → SPos
```

演習 26.3.4. [★★]　plus の変種で、上のものと型注釈を除いて同一で、型 SZero→SZero→SZero を持つ関数を書け。また、型 SPos→SNat→SPos を持つ同様の関数を書け。 □

直前の例と演習は興味深い問題を提起する。plus の変種を異なる名前で幾つも書き、引数に期待される型に基づいてどれを使うか決めなければならないといったことは、明らかに望ましくないのである。型がすべての可能性を含むような、**単一の**、例えば次のような plus が必要である。

```
  plus :   SZero→SZero→SZero
         ∧ SNat→SPos→SPos
         ∧ SPos→SNat→SPos
         ∧ SNat→SNat→SNat
```

ここで **t : S∧T** は「t は型 S と T の**両方**を持つ」という意味である。この種のオーバーロードをサポートしたいという願いから**交差型**（15.7 節）と有界量化を組み合わせた型システムの研究が生まれた。Pierce [369] を参照せよ。

演習 26.3.5. [推奨, ★★]　SNat や類似のモデルを真似て、Church ブール値（273 ページ）の型 CBool を型 SBool と二つの部分型 STrue と SFalse に一般化せよ。また、型 SFalse→STrue を持つ関数 notft と、型 STrue→SFalse を持つ同様の関数 nottf を書け。 □

演習 26.3.6. [★ ↛]　本章の導入部で、$F_{<:}$ よりも単純かつ直交するやり方で部分型付けと多相性を組み合わせられることを述べた。ここで、System F（おそらくレコードなどで拡張したもの）か

ら始め、さらに部分型関係を加え（部分型付けを伴う単純型付きラムダ計算と同様)、しかし量化は有界なものには**しないでおく**。部分型関係を、次のような通常の量化子の本体に対する共変の部分型付け規則のみで拡張する。

$$\frac{\mathsf{S <: T}}{\forall \mathsf{X.S <: \forall X.T}}$$

より単純なこの型システムでは、本章におけるどの例を定式化できるだろうか。 □

26.4 安全性

型保存の性質はカーネルとフル両方の $F_{<:}$ でかなり直接的に証明できる。本節ではカーネル $F_{<:}$ の証明を与える。フル $F_{<:}$ における議論も全く同様である。しかしながら、第 28 章で部分型付けと型検査のアルゴリズムを考えると、これら二つの変種は、本章での基本的な議論が与える示唆よりもさらに異なることがわかるだろう。多くの点において、フルの型システムはカーネルの型システムよりも解析がかなり複雑であり、またフルの型システムにはカーネルの型システムが持つ有用な性質が（決定可能な型検査さえ）欠けていることを発見するだろう。

まず型付けと部分型関係に関する幾つかの技術的事実の準備から始める。証明は導出に関する定型的な帰納法による。

補題 26.4.1.［並べ替え］ Γ を正しい形式[†1]の文脈とし、Δ は Γ の並べ替えとする。すなわち、Δ は Γ と同じ束縛を持ち、さらに Δ における順序は Γ 中の型変数のスコープを次の意味において保存しているとする。もし Γ 中のある束縛が型変数を導入し、それがさらに右で別の束縛において言及されているならば、これらの束縛は Δ に同じ順序で現れる。

(1) $\Gamma \vdash \mathsf{t : T}$ ならば $\Delta \vdash \mathsf{t : T}$

(2) $\Gamma \vdash \mathsf{S <: T}$ ならば $\Delta \vdash \mathsf{S <: T}$ □

補題 26.4.2.［弱化］

(1) $\Gamma \vdash \mathsf{t : T}$ かつ $\Gamma, \mathsf{x{:}U}$ が正しい形式ならば、$\Gamma, \mathsf{x{:}U} \vdash \mathsf{t : T}$

(2) $\Gamma \vdash \mathsf{t : T}$ かつ $\Gamma, \mathsf{X{<:}U}$ が正しい形式ならば、$\Gamma, \mathsf{X{<:}U} \vdash \mathsf{t : T}$

(3) $\Gamma \vdash \mathsf{S <: T}$ かつ $\Gamma, \mathsf{x{:}U}$ が正しい形式ならば、$\Gamma, \mathsf{x{:}U} \vdash \mathsf{S <: T}$

(4) $\Gamma \vdash \mathsf{S <: T}$ かつ $\Gamma, \mathsf{X{<:}U}$ が正しい形式ならば、$\Gamma, \mathsf{X{<:}U} \vdash \mathsf{S <: T}$ □

演習 26.4.3.［★］ 弱化の証明のどこが並べ替えに依存しているか。 □

補題 26.4.4.［部分型付け導出における項変数に関する強化］ $\Gamma, \mathsf{x{:}T}, \Delta \vdash \mathsf{S <: T}$ ならば、$\Gamma, \Delta \vdash \mathsf{S <: T}$ □

［証明］ 自明である。型付けの仮定は部分型付けの導出において何の役割も果たさない。 □

これまで通り、型保存の証明は、型付けと部分型関係を代入に関連付ける幾つかの補題に依存している。

[†1] （訳注）正しいスコープを持つこと（307 ページ参照）。

補題 26.4.5.［狭化］

(1) $\Gamma, X{<:}Q, \Delta \vdash S <: T$ かつ $\Gamma \vdash P <: Q$ ならば、$\Gamma, X{<:}P, \Delta \vdash S <: T$

(2) $\Gamma, X{<:}Q, \Delta \vdash t : T$ かつ $\Gamma \vdash P <: Q$ ならば、$\Gamma, X{<:}P, \Delta \vdash t : T$

これらの性質は、変数 X の動く範囲を制限する（狭くする）ため、しばしば**狭化**と呼ばれる。 □

［証明］ 演習［★］。 □

次に、代入と型付け関係を関連付けるお決まりの補題を示す。

補題 26.4.6.［代入は型付けを保存］ $\Gamma, x{:}Q, \Delta \vdash t : T$ かつ $\Gamma \vdash q : Q$ ならば、$\Gamma, \Delta \vdash [x \mapsto q]t : T$ □

［証明］ 上に挙げた性質を用いた、$\Gamma, x{:}Q, \Delta \vdash t : T$ の導出に関する帰納法による。 □

簡約の途中で型変数に型を代入するかもしれないため、型の代入と型付けを関連付ける補題も必要である。その補題の証明（特に、T-Sub の場合）は、代入と部分型付けを関連付ける新しい補題に依存している。

定義 26.4.7. Γ 中のすべての束縛の右側の X に S を代入して得られた文脈を $[X \mapsto S]\Gamma$ と書く。 □

補題 26.4.8.［型代入は部分型付けを保存］ $\Gamma, X{<:}Q, \Delta \vdash S <: T$ かつ $\Gamma \vdash P <: Q$ ならば、$\Gamma, [X \mapsto P]\Delta \vdash [X \mapsto P]S <: [X \mapsto P]T$ □

文脈のうち X の束縛に続く部分でのみ X を代入する必要があることに注意されたい。なぜなら、スコープの扱いの慣習により X の束縛より左の型は X を含まないためである。

［証明］ $\Gamma, X{<:}Q, \Delta \vdash S <: T$ の導出に関する帰納法による。興味深いのは次の二つのみである。

S-TVar の場合 $S = Y \quad Y{<:}T \in (\Gamma, X{<:}Q, \Delta)$

さらに二つの場合に分ける必要がある。$Y \neq X$ ならば、結論は S-TVar より直ちに得られる。一方、$Y = X$ ならば、$T = Q$ かつ $[X \mapsto P]S = P$ である。前提 $\Gamma \vdash P <: Q$ より、求めていた結果が得られる。

S-All の場合 $S = \forall Z{<:}U_1.S_2 \qquad T = \forall Z{<:}U_1.T_2$

$\Gamma, X{<:}Q, \Delta, Z{<:}U_1 \vdash S_2 <: T_2$

帰納法の仮定より、$\Gamma, [X \mapsto P]\Delta, Z{<:}[X \mapsto P]U_1 \vdash [X \mapsto P]S_2 <: [X \mapsto P]T_2$。S-All より、$\Gamma, [X \mapsto P]\Delta \vdash \forall Z{<:}[X \mapsto P]U_1.[X \mapsto P]S_2 <: \forall Z{<:}[X \mapsto P]U_1.[X \mapsto P]T_2$、すなわち、所要の通り、$\Gamma, [X \mapsto P]\Delta \vdash [X \mapsto P](\forall Z{<:}U_1.S_2) <: [X \mapsto P](\forall Z{<:}U_1.T_2)$ が得られる。 □

同様の補題が型代入と型付けを関連付ける。

補題 26.4.9.［型代入は型付けを保存］ $\Gamma, X{<:}Q, \Delta \vdash t : T$ かつ $\Gamma \vdash P <: Q$ ならば、$\Gamma, [X \mapsto P]\Delta \vdash [X \mapsto P]t : [X \mapsto P]T$ □

［証明］ $\Gamma, X{<:}Q, \Delta \vdash t : T$ の導出に関する帰納法による。興味深い場合のみ示す。

T-TApp の場合 $t = t_1\ [T_2] \qquad \Gamma, X{<:}Q, \Delta \vdash t_1 : \forall Z{<:}T_{11}.T_{12}$

$T = [Z \mapsto T_2]T_{12}$

帰納法の仮定より、$\Gamma, [X \mapsto P]\Delta \vdash [X \mapsto P]t_1 : [X \mapsto P](\forall Z{<:}T_{11}.T_{12})$、すなわち、$\Gamma, [X \mapsto P]\Delta \vdash$

$[X \mapsto P]t_1 : \forall Z<:T_{11}.[X \mapsto P]T_{12}$。T-TAPP より、$\Gamma, [X \mapsto P]\Delta \vdash [X \mapsto P]t_1\ [[X \mapsto P]T_2] : [Z \mapsto [X \mapsto P]T_2]([X \mapsto P]T_{12})$、すなわち、$\Gamma, [X \mapsto P]\Delta \vdash [X \mapsto P](t_1\ [T_2]) : [X \mapsto P]([Z \mapsto T_2]T_{12})$。

T-SUB の場合 $\Gamma, X<:Q, \Delta \vdash t : S \quad \Gamma, X<:Q, \Delta \vdash S <: T$

帰納法の仮定より、$\Gamma, [X \mapsto P]\Delta \vdash [X \mapsto P]t : [X \mapsto P]S$。代入における部分型付けの保存（補題 26.4.8）により、$\Gamma, [X \mapsto P]\Delta \vdash [X \mapsto P]S <: [X \mapsto P]T$ が得られ、T-SUB より結論を導ける。

□

次に、部分型付けに関する簡単な構造的性質を証明する。

補題 26.4.10.［部分型関係の逆転、右から左］

(1) $\Gamma \vdash S <: X$ ならば、S は型変数である。
(2) $\Gamma \vdash S <: T_1 \to T_2$ ならば、S は型変数であるかまたは S は $S_1 \to S_2$ であり、ここで $\Gamma \vdash T_1 <: S_1$ かつ $\Gamma \vdash S_2 <: T_2$ である。
(3) $\Gamma \vdash S <: \forall X<:U_1.T_2$ ならば、S は型変数であるかまたは S は $\forall X<:U_1.S_2$ であり、ここで $\Gamma, X<:U_1 \vdash S_2 <: T_2$ である。 □

［証明］ (1) は部分型付け導出に関する簡単な帰納法による。興味深いのは規則 S-TRANS の場合のみであり、これは二つの帰納法の仮定を使って進む。一つ目は右の前提、二つ目は左の前提に対してである。残りの議論も同様であり、(1) を推移律の場合に使う。 □

演習 26.4.11.［推奨, ★★］ 次の性質「左から右への逆転」を示せ。

(1) $\Gamma \vdash S_1 \to S_2 <: T$ ならば、$T = Top$ または $T = T_1 \to T_2$ であり、ここで $\Gamma \vdash T_1 <: S_1$ かつ $\Gamma \vdash S_2 <: T_2$ である。
(2) $\Gamma \vdash \forall X<:U.S_2 <: T$ ならば、$T = Top$ または $T = \forall X<:U.T_2$ であり、ここで $\Gamma, X<:U \vdash S_2 <: T_2$ である。
(3) $\Gamma \vdash X <: T$ ならば、$T = Top$ または $T = X$ または $\Gamma \vdash S <: T$ のいずれかであり、ここで $X<:S \in \Gamma$ である。
(4) $\Gamma \vdash Top <: T$ ならば、$T = Top$ である。 □

次の補題 26.4.10 は、型付け関係の単純な構造的性質を証明するために用いられる。この性質は型保存の証明における重要な場合分けにおいて必要となる。

補題 26.4.12.

(1) $\Gamma \vdash \lambda x:S_1.s_2 : T$ かつ $\Gamma \vdash T <: U_1 \to U_2$ ならば、$\Gamma \vdash U_1 <: S_1$ かつ、ある S_2 が存在し $\Gamma, x:S_1 \vdash s_2 : S_2$ かつ $\Gamma \vdash S_2 <: U_2$ である。
(2) $\Gamma \vdash \lambda X<:S_1.s_2 : T$ かつ $\Gamma \vdash T <: \forall X<:U_1.U_2$ ならば、$U_1 = S_1$ かつ、ある S_2 が存在し $\Gamma, X<:S_1 \vdash s_2 : S_2$ かつ $\Gamma, X<:S_1 \vdash S_2 <: U_2$ である。 □

［証明］ 型付け導出に関する単純な帰納法により、帰納ケース（規則 T-SUB）では補題 26.4.10 を用いる。 □

これらの事実を手にすれば、型保存の証明は単純である。

定理 26.4.13.［保存］ $\Gamma \vdash t : T$ かつ $t \longrightarrow t'$ ならば、$\Gamma \vdash t' : T$ である。 □

［証明］ $\Gamma \vdash t : T$ の導出に関する帰納法による。すべての場合は単純であり、上で証明した事実を

用いる。

T-VAR、T-ABS、T-TABS の場合　$t = x$、 $t = \lambda x{:}T_1.t_2$、 または $t = \lambda X{<:}U.t$
　$t \longrightarrow t'$ を仮定しており、かつ変数、ラムダ抽象、型抽象に関する評価規則がないため、これらの場合は実際には起こりえない。

T-APP の場合　$t = t_1\ t_2$　　$\Gamma \vdash t_1 : T_{11} \to T_{12}$　　$T = T_{12}$　　$\Gamma \vdash t_2 : T_{11}$
　評価関係の定義より、次の三つに場合分けされる。

　　$t_1 \longrightarrow t_1'$　　$t' = t_1'\ t_2$ の場合
　　　このとき結論は帰納法の仮定と T-APP より得られる。

　　t_1 が値　　$t_2 \longrightarrow t_2'$　　$t' = t_1\ t_2'$ の場合
　　　同様。

　　$t_1 = \lambda x{:}U_{11}.u_{12}$　　$t' = [x \mapsto t_2]u_{12}$ の場合
　　　補題 26.4.12 より、$\Gamma, x{:}U_{11} \vdash u_{12} : U_{12}$ がある U_{12} について成立し、さらに $\Gamma \vdash T_{11} <: U_{11}$ かつ $\Gamma \vdash U_{12} <: T_{12}$ である。代入における型付けの保存（補題 26.4.6）より、$\Gamma \vdash [x \mapsto t_2]u_{12} : U_{12}$ であり、これより、T-SUB を用いて $\Gamma \vdash [x \mapsto t_2]u_{12} : T_{12}$ が得られる。

T-TAPP の場合　$t = t_1\ [T_2]$　　$\Gamma \vdash t : \forall X{<:}T_{11}.T_{12}$
　　　　　　　$T = [X \mapsto T_2]T_{12}$　$\Gamma \vdash T_2 <: T_{11}$
　評価関係の定義より、次の二つに場合分けされる。

　　$t_1 \longrightarrow t_1'$　　$t' = t_1'\ [T_2]$ の場合
　　　結論は帰納法の仮定と T-TAPP より得られる。

　　$t_1 = \lambda X{<:}U_{11}.u_{12}$　　$t' = [X \mapsto T_2]u_{12}$ の場合
　　　補題 26.4.12 より、$U_{11} = T_{11}$ かつ $\Gamma, X{<:}U_{11} \vdash u_{12} : U_{12}$ であり、ここで $\Gamma, X{<:}U_{11} \vdash U_{12} <: T_{12}$。代入における型付けの保存（補題 26.4.9）より、$\Gamma \vdash [X \mapsto T_2]u_{12} : [X \mapsto T_2]U_{12}$ であり、これより $\Gamma \vdash [X \mapsto T_2]u_{12} : [X \mapsto T_2]T_{12}$ が補題 26.4.8 と T-SUB により得られる。

T-SUB の場合　$\Gamma \vdash t : S$　　$\Gamma \vdash S <: T$
　帰納法の仮定より、$\Gamma \vdash t' : S$ であり、結論は T-SUB より導かれる。 □

$F_{<:}$ の進行定理は部分型付けを伴う単純型付きラムダ計算のときの素直な拡張である。いつものように、まずは矢印型と量化型の閉じた値が取りうる形を教えてくれる標準形の性質を記す。

補題 26.4.14.［標準形］

(1) v が型 $T_1 \to T_2$ を持つ閉じた値ならば、v は $\lambda x{:}S_1.t_2$ の形を持つ。
(2) v が型 $\forall X{<:}T_1.T_2$ を持つ閉じた値ならば、v は $\lambda X{<:}T_1.t_2$ の形を持つ。 □

［証明］ 両方とも型付け導出に関する帰納法により導く。ここでは (2) のみ議論する。型付け規則を調べると、$\vdash v : \forall X{<:}T_1.T_2$ の導出に用いる最後の規則は T-TABS か T-SUB のどちらかである。T-TABS のとき、求めたい結論は規則の前提より直ちに得られる。このため最後の規則が T-SUB であると仮定する。規則の前提より、$\vdash v : S$ かつ $S <: \forall X{<:}T_1.T_2$ が得られる。逆転補題（補題 26.4.10）より、S が $\forall X{<:}T_1.S_2$ の形であることがわかる。ここで結論が帰納法の仮定より導かれる。 □

これを手にすれば、進行の証明は単純である。

定理 26.4.15.［進行］ t が閉じた、正しく型付けされた $F_{<:}$ の項ならば、t は値であるか、ある t' が存在し $t \longrightarrow t'$ である。 □

→ ∀ <: Top ∃ 　　　　$F_{<:}$（図 26-1）と有界でない存在型（図 24-1）の拡張

新しい構文形式

T ::= ...	型：
{∃X<:T,T}	存在型

新しい部分型付け規則 　　$\Gamma \vdash S <: T$

$$\frac{\Gamma, X{<:}U \vdash S_2 <: T_2}{\Gamma \vdash \{\exists X{<:}U, S_2\} <: \{\exists X{<:}U, T_2\}} \quad \text{(S-SOME)}$$

新しい型付け規則 　　$\Gamma \vdash t : T$

$$\frac{\Gamma \vdash t_2 : [X \mapsto U]T_2 \qquad \Gamma \vdash U <: T_1}{\Gamma \vdash \{{*}U, t_2\} \text{ as } \{\exists X{<:}T_1, T_2\} : \{\exists X{<:}T_1, T_2\}} \quad \text{(T-PACK)}$$

$$\frac{\Gamma \vdash t_1 : \{\exists X{<:}T_{11}, T_{12}\} \qquad \Gamma, X{<:}T_{11}, x{:}T_{12} \vdash t_2 : T_2}{\Gamma \vdash \text{let } \{X,x\}{=}t_1 \text{ in } t_2 : T_2} \quad \text{(T-UNPACK)}$$

▲ 図 26-3. 有界存在量化（カーネル版）

［証明］ 型付け導出に関する帰納法による。t が閉じているため、変数の場合は出現しえない。二種類のラムダ抽象の場合は自明である。なぜなら項のラムダ抽象と型のラムダ抽象は両方とも値だからである。項の適用、型適用、包摂の場合はもっと興味深い。ここでは後者二つを示す（項の適用は型適用と同様である）。

T-TAPP の場合 　$t = t_1\ [T_2]$ 　$\vdash t_1 : \forall X{<:}T_{11}.T_{12}$

$\Gamma \vdash T_2 <: T_{11}$ 　$T = [X \mapsto T_2]T_{12}$

帰納法の仮定より、t_1 は値であるか、1 ステップ簡約できるかのどちらかである。t_1 が 1 ステップ簡約できるならば、規則 E-TAPP が t に適用できる。さもなくば、t_1 が値のとき、標準形補題（補題 26.4.14）の (2) より t_1 は $\lambda X{<:}T_{11}.t_{12}$ の形であるため、規則 E-TAPPTABS が t に適用できる。

T-SUB の場合 　$\Gamma \vdash t : S$ 　$\Gamma \vdash S <: T$

帰納法の仮定より直ちに結論が導かれる。 □

演習 26.4.16. [★★★ ↛] この節の議論をフル $F_{<:}$ に拡張せよ。 □

26.5 有界存在型

ちょうど全称型に行ったのと同様に、存在型（第 24 章）にも上界を加えることができ、**有界存在型**が図 26-3 のように得られる。部分型付け規則 S-SOME も有界全称型と同様に二通りあり、一方では比較される量化子が同一でなければならないが、他方では異なってもよい。

演習 26.5.1. [★] S-SOME の**フル**版はどんなものか。 □

演習 26.5.2. [★] レコードと存在型がある純粋な System F（ただし部分型付けはない）において、

```
{*Nat, {a=5,b=7}} as T;
```

を正しく型付けする T を選ぶ方法は幾つあるか。部分型付けと有界存在型を加えると、より多くの方法が得られるか。 □

289 ページでは普通の存在型を用いて抽象データ型を実装するやり方を見た。存在量化子に上界を加えると、それに伴う詳細化が ADT レベルで得られる。これは Cardelli and Wegner [88] によ

り**半抽象型**と名付けられた。鍵となる直観は、有界存在量化によって表現型に関する**一部の**構造が外の世界に露出される一方で、その表現型の正確な正体は隠れたままになることである。

例えば、289 ページのようにカウンタの ADT を実装するが、ここで型注釈に上界 Counter<:Nat を加えたとしよう。

```
counterADT =
   {*Nat, {new = 1, get = λi:Nat. i, inc = λi:Nat. succ(i)}}
   as {∃Counter<:Nat,
       {new: Counter, get: Counter→Nat, inc: Counter→Counter}};
▸ counterADT : {∃Counter<:Nat,
                {new:Counter,get:Counter→Nat,inc:Counter→Counter}}
```

このカウンタ ADT は型と項の部品を変数 Counter と counter に束縛し、counter のフィールドを用いてカウンタ上の操作を実行するなど、これまでと全く同じように利用できる。

```
let {Counter,counter} = counterADT in
counter.get (counter.inc (counter.inc counter.new));
▸ 3 : Nat
```

さらに、Counter の値は数として直接使える。

```
let {Counter,counter} = counterADT in
succ (succ (counter.inc counter.new));
▸ 4 : Nat
```

一方で、依然として数は Counter として使えない。

```
let {Counter,counter} = counterADT in
counter.inc 3;
▸ Error: parameter type mismatch
```

結果的に、今回のカウンタの抽象化では、表現を暴露することによりカウンタを外の世界から使いやすくしたが、一方でカウンタの作り方に関する統制は保たれている。

演習 26.5.3. [★★★] 次のような二つの抽象データ型 Counter と ResetCounter を定義したいとしよう。(1) 両方の ADT が new、get、inc の操作を提供し、(2) それに加え ResetCounter は、カウンタを取りある固定値（例えば 1）にセットされた新しいカウンタを返す reset 操作を提供し、(3) 二つの ADT の利用者は Counter の代わりに ResetCounter を使うことができ（つまり ResetCounter <: Counter)、(4) それ以上はカウンタとリセットカウンタがどう表現されているかについて利用者に暴露されないとする。これは有界存在型パッケージを使って達成できるだろうか。 □

291 ページでの、存在型によるオブジェクトの表現も、同様に洗練できる。以前は、存在型パッケージの証人型はオブジェクトの内部状態の型を表現するために用いられ、その内部状態とはインスタンス変数レコードだった。有界でない存在型の代わりに有界存在型を用いれば、オブジェクトのインスタンス変数の名前や型の、全体でなく一部を、外の世界に暴露できる。例えば、次は一部だけ可視な内部状態を持つカウンタオブジェクトである。そのフィールド x だけが見えている一方で、（あまり面白くない）private フィールドの可視性が制限されている。

```
  c = {*{x:Nat, private:Bool},
       {state = {x=5, private=false},
        methods = {get = λs:{x:Nat}. s.x,
                   inc = λs:{x:Nat,private:Bool}.
                           {x=succ(s.x), private=s.private}}}}
      as {∃X<:{x:Nat}, {state:X, methods: {get:X→Nat, inc:X→X}}};
▶ c : {∃X<:{x:Nat}, {state:X,methods:{get:X→Nat,inc:X→X}}}
```

このカウンタの半抽象データ型を用いれば、カウンタオブジェクトの値を得るために get メソッドを呼び出すか、内部に到達し状態のフィールド x を見るかの選択肢が生まれる。

演習 26.5.4. [★★]　24.3 節の全称型を用いた存在型の表現を、有界全称型を用いた有界存在型の表現へと拡張する方法を示せ。部分型付け規則 S-SOME が、その表現と有界全称型の部分型付け規則から導けることを確かめよ。 □

26.6 注記

CLU（Liskov et al. [289; 288]; Schaffert [425]; Scheifler [426]）は型安全な有界量化を持つ最初の言語だったようである。CLU のパラメータ境界の概念は実質的には上界そのものが量化の範囲に含まれる有界量化（26.2 節）が複数の型引数に一般化されたものである。

ここで述べた形での有界量化のアイディアは Cardelli と Wegner によって言語 Fun において導入された [88]。彼らの「Kernel Fun」計算は我々のカーネル $F_{<:}$ に対応する。Cardelli による初期の非形式的なアイディアに基づき、Mitchell [318] によって開発された技法を用いて形式化され、Fun は Girard-Reynolds 多相（Girard [177]; Reynolds [408]）と Cardelli の部分型付けに関する一階の計算 [73] を統合した。最初の Fun は Bruce and Longo [64]、また Curien and Ghelli [127] により単純化および少し一般化され、我々がフル $F_{<:}$ と呼ぶ計算が生まれた。有界量化に関する最も包括的な論文は Cardelli、Martini、Mitchell、Scedrov によるサーベイ [86] である。

$F_{<:}$ とその同類はプログラミング言語の理論家と設計者によって広く研究された。Cardelli と Wegner のサーベイ論文により有界量化を用いた最初のプログラミング例が示され、より多くの例が Cardelli の冪カインドの研究 [76] で開発された。Curien and Ghelli [127; 171] は $F_{<:}$ に関する多くの構文的性質について述べている。密接に関連する型システムの意味論的側面は Bruce and Longo [64]、Martini [302]、Breazu-Tannen, Coquand, Gunter, and Scedrov [57]、Cardone [89]、Cardelli and Longo [85]、Cardelli, Martini, Mitchell, and Scedrov [86]、Curien and Ghelli [127; 126]、そして Bruce and Mitchell [65] において研究された。$F_{<:}$ は、Cardelli and Mitchell [87]、Bruce [58]、Cardelli [80]、Canning, Cook, Hill, Olthoff, and Mitchell [71] において、レコード型と継承に関する、より豊富な概念を含むように拡張された。有界量化は Cardelli のプログラミング言語 Quest（Cardelli [79]; Cardelli and Longo [85]）、HP 研究所で開発された Abel 言語（Canning, Cook, Hill, and Olthoff [70]; Canning, Cook, Hill, Olthoff, and Mitchell [71]; Canning, Hill, and Olthoff [72]; Cook, Hill, and Canning [114]）、より最近の GJ（Bracha, Odersky, Stoutamire, and Wadler [54]）、Pict（Pierce and Turner [376]）、Funnel（Odersky [341]）の設計において重要な役割を果たしている[†2]。

代数的データ型の Church 表現における有界量化の効果（312 ページ）は Ghelli [171] と Cardelli, Martini, Mitchell, and Scedrov [86] によって考察された。

交差型（15.7 節）による $F_{<:}$ の拡張は Pierce [365; 369] により研究された。高階のカインドを伴

†2（訳注）より最近の言語としては、C#、Java、Scala などに含まれている。

う変種は Compagnoni and Pierce [108] により多重継承を持つオブジェクト指向言語のモデル化へと応用された。そのメタ理論的性質は Compagnoni [107] で解析された。

第 27 章

事例：命令的オブジェクト再考

本章の例はレコード（図 15-3）と参照（図 13-1）を持つ $F_{<:}$ の項である。関連する OCaml 実装は `fullfsubref` である。

第 18 章では、レコード、参照、部分型を持つ単純型付き計算において、命令的オブジェクト指向プログラミングの核心部分をモデル化するような複数のイディオムを説明した。第 18 章の終わり（18.12 節）では、そのオブジェクトの実行時効率を改善するため、オブジェクトのメソッドテーブルの構築を、メソッド呼び出し時ではなくオブジェクト生成時に行うようにした。本章では有界量化を用いて、このモデルの効率をさらに改善する[†1]。

18.12 節における鍵となるアイディアは、「`self` メソッドテーブル」への**参照**をクラス呼び出し時にクラスへ渡すことだった。クラスは自身のメソッドを定義するときにこの参照を用い、その後この参照はクラスから返される完全なメソッドテーブルを指すように張り直される。例えば、`SetCounter` と `CounterRep` がメソッド `get`、`set`、`inc` を持つカウンタオブジェクトのクラスの、公開インターフェイスと内部表現の型であるとする。

```
SetCounter = {get:Unit→Nat, set:Nat→Unit, inc:Unit→Unit};

CounterRep = {x: Ref Nat};
```

すると、セットカウンタのクラスを次のように実装できる。

```
setCounterClass =
  λr:CounterRep. λself: Source SetCounter.
    {get = λ_:Unit. !(r.x),
     set = λi:Nat. r.x:=i,
     inc = λ_:Unit. (!self).set
                      (succ ((!self).get unit))};
▸ setCounterClass : CounterRep → (Source SetCounter) → SetCounter
```

引数 `self` の型として `Ref SetCounter` ではなく `Source SetCounter` を使う。その理由は `setCounterClass` のサブクラスを定義するとき、その新しいクラスの `self` は別の型を持つことになるからである。例えば、`InstrCounter` と `InstrCounterRep` が計器付きカウンタオブジェクトのクラスの、インターフェイスと表現型であるとする。

---

†1 (訳注：原著正誤表より) 本章については以下の問題が指摘されている（以下、原著者のコメントを翻訳の上、転載する)。

$F_{<:}$ を使う理由は、メソッドテーブルをオブジェクトごとに作らず、クラスごとに作るだけで済ませることにあったはずである。実際に本章のコードは「何か」の生成を各クラス一回だけに抑えることができているものの、その「何か」は**関数**であって、その関数は **`self` を介したオープンな再帰呼び出しのたびに**メソッドテーブルを作ってしまう（メソッド呼び出しに伴い `(!self r)` が呼ばれることを見よ)。言い換えると、本章の最終的な効率は第 18 章の最後で達成したものよりも悪くなっている。

この問題を指摘してくれた John Tang Boyland に大いに感謝する。加えて John は別の事例として、オブジェクトを命令的スタイルで実装するために、第 32 章に倣って $F^{\omega}_{<:}$ を用いることを提案してくれた。この構成の概要は以下の場所にある。

```
http://www.cis.upenn.edu/~bcpierce/tapl/boyland-object-encoding.txt
```

また、このアイディアを実装したコードが以下の場所にある。

```
http://www.cs.uwm.edu/classes/cs790/types-f2003/Chapter27-replacement.txt
http://www.cs.uwm.edu/classes/cs790/types-f2003/fullfomsubref.tar.gz
```

```
InstrCounter = {get:Unit→Nat, set:Nat→Unit,
                inc:Unit→Unit, accesses:Unit→Nat};

InstrCounterRep = {x: Ref Nat, a: Ref Nat};
```

すると、クラスは次のように定義できる。

```
instrCounterClass =
  λr:InstrCounterRep. λself: Source InstrCounter.
  let super = setCounterClass r self in
  {get = super.get,
   set = λi:Nat. (r.a:=succ(!(r.a)); super.set i),
   inc = super.inc,
   accesses = λ_:Unit.
               !(r.a)};
▸ instrCounterClass : InstrCounterRep →
                      (Source InstrCounter) → InstrCounter
```

ここで引数 **self** の型は **Source InstrCounter** である。**super** を構築するときに、この **self** を **setCounterClass** の引数 **self** として渡すためには、**Source InstrCounter** から **Source SetCounter** への型強制ができる必要がある。共変な構築子 **Source** はこの型強制を許すが、非変な構築子 **Ref** はこれを許さない。

しかし 18.12 節の終わりで見たように、クラスのこのモデルの効率はまだ最適ではない。与えられた一つのクラスからインスタンス化されたすべてのオブジェクトには同一のメソッドテーブルが関連付けられているから、このテーブルを構築するのは**クラス**生成時の一度だけで済むこと、そしてオブジェクトが生成されるときはいつもそのテーブルを再利用できることが望ましい。そうすれば現実のオブジェクト指向言語の実装方式がより正確に反映されるだろう。その慣習では、オブジェクトはメソッドを一切持たず、クラスを表すデータ構造へのポインタのみを持ち、メソッドは実際にはそのデータ構造に格納される[†2]。

同じことの別の説明としては、上のクラスで、引数の順番（まずインスタンス変数、次に **self**）が逆向きであることを見るとよい。引数 **self** はメソッドテーブルを構築するのに必要だが、インスタンス変数レコード **r** はメソッドが実際に呼ばれるまで使われていない。もし **self** が最初の引数であったならば、引数 **r** が渡される前にメソッドテーブルを計算できる。つまりクラスを一度引数 **self** へ部分適用し、この計算を一度だけ実行し、結果であるメソッドテーブルを複数のインスタンス変数レコードへ適用することで、メソッドテーブルの複数のコピーを作れる。具体的には **setCounterClass** を次のように書き換えたい。

[†2] 実際には、現実のオブジェクト指向言語はもう一歩進んでいる。完全なメソッドテーブルを計算し格納する代わりに、各クラスはスーパークラスに対して加えたりオーバーライドしたりするメソッドだけを格納している。つまり我々が考えるようなメソッドテーブルが構築されることはない。メソッド呼び出し時にはレシーバオブジェクトの実際のクラスを起点に、求めるメソッドの定義が見つかるまで単純にクラス階層を上へ辿る。（(訳注：原著正誤表より) メソッド探索についてのこの説明は簡潔すぎであり、誤解を招くかもしれない。実際の実装はずっと複雑なことを行い、効率を上げている。）この種の実行時探索は静的型解析に厄介な問題をもたらす。ここではそれは扱わない。

```
setCounterClass =
  λself: Source (CounterRep→SetCounter).
  λr:CounterRep.
    {get = λ_:Unit. !(r.x),
     set = λi:Nat. r.x:=i,
     inc = λ_:Unit. (!self r).set
                        (succ ((!self r).get unit))};
▸ setCounterClass : (Source (CounterRep→SetCounter)) →
                     CounterRep → SetCounter
```

この setCounterClass と前の setCounterClass には三つの明確な違いがある。一つ目には、新しい setCounterClass は r の前に self を受け取る。二つ目には、self の型が SetCounter から CounterRep→SetCounter に変わる。この変更は一つ目の変更により強制される。なぜならば、self の型はクラスによって返されるメソッドテーブルの型と同じでなければならないからである。三つ目には、クラスの本体において !self を使うところはすべて (!self r) になる。この変更は二つ目の変更により強制される。

新しいカウンタのためのインスタンス化関数は次のように定義される。

```
newSetCounter =
  let m = ref (λr:CounterRep. error as SetCounter) in
  let m' = setCounterClass m in
  (m := m';
   λ_:Unit. let r = {x=ref 1} in m' r);
▸ newSetCounter : Unit → SetCounter
```

定義の最初の三行が、newSetCounter が定義されるときに一度だけ評価されることに注意されたい。評価は最後の行にある自明なラムダ抽象で停止し、このラムダ抽象をオブジェクト生成のためにくり返し適用できる。この適用は毎回新たなインスタンス変数レコード r のために記憶域を確保し、メソッドテーブル m' を r で具体化し、新たなオブジェクトを得る。

残念なことに、setCounterClass にこのような変更を加えることで、状態の型 CounterRep の**反変な**出現が導入されてしまう。これは setCounterClass のサブクラスを定義しようとするときに我々を悩ませることになる。

```
instrCounterClass =
  λself: Source (InstrCounterRep→InstrCounter).
  let super = setCounterClass self in
  λr:InstrCounterRep.
  {get = (super r).get,
   set = λi:Nat. (r.a:=succ(!(r.a)); (super r).set i),
   inc = (super r).inc,
   accesses = λ_:Unit. !(r.a)};
▸ Error: parameter type mismatch
```

不適合は super の定義で起こる。そこではサブクラスへ渡されたのと同じ self によってスーパークラス setCounterClass がインスタンス化される。残念なことに現在の self は型 InstrCounterRep→InstrCounter を持つ関数（への参照）であり、この型は CounterRep→SetCounter の部分型ではない。矢印の左側があべこべなためである。

setCounterClass をもう一度書き換えることでこの困難に対処できる。今回は有界量化を使う。

```
setCounterClass =
  λR<:CounterRep.
  λself: Source (R→SetCounter).
    λr: R.
       {get = λ_:Unit. !(r.x),
        set = λi:Nat.  r.x:=i,
        inc = λ_:Unit. (!self r).set (succ((!self r).get unit))};
▸ setCounterClass : ∀R<:CounterRep.
                        (Source (R→SetCounter)) → R → SetCounter
```

この変更により、引数 self に対しての setCounterClass の要求が少し減る。少し擬人化すると、以前の setCounterClass は環境に対して、「CounterRep を引数として受け付ける引数 self を渡してください。私はそれを使って、同じく CounterRep を引数として期待するようなメソッドテーブルを構築します」と言うものであると特徴付けられる。新しい setCounterClass は代わりに、「これから構築するオブジェクトのための表現の型 R を教えてください。その型は少なくともフィールド x を持っていなければなりません。私がそれを使う必要があるからです。そして、R を引数として受け付け、少なくとも SetCounter インターフェイスのメソッドを持つメソッドテーブルを返すような、self（への参照）を与えてください。すると私は同様のメソッドテーブルをもう一つ構築し、返します」と言う。

この変更の影響はサブクラス instrCounterClass を定義するときに最も明瞭である。

```
instrCounterClass =
  λR<:InstrCounterRep.
  λself: Source (R→InstrCounter).
    λr: R.
      let super = setCounterClass [R] self in
         {get = (super r).get,
          set = λi:Nat. (r.a:=succ(!(r.a)); (super r).set i),
          inc = (super r).inc,
          accesses = λ_:Unit. !(r.a)};
▸ instrCounterClass : ∀R<:InstrCounterRep.
                         (Source (R→InstrCounter)) →
                         R → InstrCounter
```

前の定義が失敗しこの定義がうまくいく理由は、super に束縛された式において包摂が異なる使われ方をすることによる。ここでは、self の型を、スーパークラスが受け付けられるように昇格させた。これまでは次の関係式を示そうとしていたが、

$$\begin{array}{ll} & \text{Source(InstrCounterRep}\to\text{InstrCounter)} \\ <: & \text{Source(CounterRep}\to\text{SetCounter)} \end{array}$$

これは偽であった。今回は次を示せばよい。

$$\text{Source(R}\to\text{InstrCounter)} <: \text{Source(R}\to\text{SetCounter)}$$

これは真である。

クラスの新しい表現のためのオブジェクト生成関数は、以前のものとよく似ている。例えば、次は計器付きカウンタのサブクラスに対する生成関数である。

```
newInstrCounter =
  let m = ref (λr:InstrCounterRep. error as InstrCounter) in
  let m' = instrCounterClass [InstrCounterRep] m in
  (m := m';
   λ_:Unit. let r = {x=ref 1, a=ref 0} in m' r);
▸ newInstrCounter : Unit → InstrCounter
```

唯一の違いは instrCounterClass を、インスタンス変数レコードの実際の型 InstrCounterRep で具体化する必要があることである。以前と同じく、最初の三行は newInstrCounter の束縛が作られる時点で一度だけ実行される。

最後に、これらのカウンタが期待通り振る舞うことを実証するためのテストを幾つか以下に示す。

```
ic = newInstrCounter unit;
ic.inc unit;
ic.get unit;
▸ 2 : Nat

ic.accesses unit;
▸ 1 : Nat
```

演習 27.1. [推奨, ★★★]　ここでのクラスの新しい表現は、型構築子 Source の共変性に頼っている。有界量化と非変な構築子 Ref だけを持つ言語において同じ効率を達成できるか（すなわち、正しく型付けされ、同じ操作的振る舞いを持つクラスの表現を与えられるか)。　□

第 28 章

有界量化のメタ理論

本章で扱う体系は純粋な $F_{<:}$（図 26-1）である。対応する実装は purefsub で、fullfsub 実装は存在型（図 24-1）と第 11 章の幾つかの拡張も含んでいる。

本節では $F_{<:}$ の部分型付けと型検査のアルゴリズムを説明する。この型システムの変種である、カーネル型システムとフル型システムの両方を扱う。両者の振る舞いは、やや異なる。一部の性質は両方で成り立つものの、フル $F_{<:}$ では証明がより困難である。フル $F_{<:}$ では完全に失われてしまう性質もある。これらは、フル型システムの表現力の代償である。

最初に 28.1 節と 28.2 節において、両方の型システムで動作する型検査アルゴリズムを示す。次に、部分型検査について考える。まずは 28.3 節でカーネル型システムを、次に 28.4 節でフル型システムを扱う。28.5 節ではフル $F_{<:}$ における部分型付けに関して議論を続け、部分型関係が決定不能であるという驚くべき事実に注目する。28.6 節では カーネル型システムが結びと交わりを持つ一方、フル型システムはそれらを持たないことを示す。28.7 節では有界存在型により引き起こされる幾つかの問題について触れ、28.8 節では最小の Bot 型を加えることの効果について考える。

28.1 型の露出

部分型付けを持つ単純型付きラムダ計算に対する 16.2 節の型検査アルゴリズムにおいて鍵となるアイディアは、それぞれの項の**最小型**を部分項の最小型を用いて計算することにあった。基本的な考えは $F_{<:}$ にも流用できるが、この型システムにおける型変数の存在により生じる、ちょっとした複雑さを考慮に入れる必要がある。次の項について考える。

```
  f = λX<:Nat→Nat. λy:X. y 5;
▸ f : ∀X<:Nat→Nat. X → Nat
```

この項は関数適用 y 5 における変数 y の型が T-SUB により Nat→Nat に昇格でき、明らかに正しく型付けされている。しかし y の**最小**型は X であり、矢印型ではない。関数適用全体の最小型を見つけるには、y が持ちうる最小の矢印型を見つける必要がある。すなわち、型変数 X の上位型で最小の矢印型である。驚くことではないが、型変数以外の何かになるまで y の最小型を**昇格**すれば、この型が得られる。

形式的には、$\Gamma \mapsto S \Uparrow T$ と書き（読み方は「S は Γ の下で T へと露出する」)、「T は S の、最小の変数でない上位型である」ということを意味するものとする。露出は図 28-1 に示すように変数の昇格の繰り返しで定義される。

→ ∀ <: Top

露出 $\boxed{\Gamma \mapsto T \Uparrow T'}$

$$\frac{X{<:}T \in \Gamma \qquad \Gamma \mapsto T \Uparrow T'}{\Gamma \mapsto X \Uparrow T'} \quad \text{(XA-PROMOTE)}$$

$$\frac{T \text{ は型変数でない}}{\Gamma \mapsto T \Uparrow T} \quad \text{(XA-OTHER)}$$

▲ 図 28-1. $F_{<:}$ の型露出アルゴリズム

→ ∀ <: Top　　λ<:（図 16-3）の拡張

アルゴリズム的型付け　$\boxed{\Gamma \mapsto t : T}$

$$\frac{x{:}T \in \Gamma}{\Gamma \mapsto x : T} \quad \text{(TA-VAR)}$$

$$\frac{\Gamma, x{:}T_1 \mapsto t_2 : T_2}{\Gamma \mapsto \lambda x{:}T_1.t_2 : T_1 \to T_2} \quad \text{(TA-ABS)}$$

$$\frac{\Gamma \mapsto t_1 : T_1 \qquad \Gamma \mapsto T_1 \Uparrow (T_{11} \to T_{12}) \qquad \Gamma \mapsto t_2 : T_2 \qquad \Gamma \mapsto T_2 <: T_{11}}{\Gamma \mapsto t_1\ t_2 : T_{12}} \quad \text{(TA-APP)}$$

$$\frac{\Gamma, X{<:}T_1 \mapsto t_2 : T_2}{\Gamma \mapsto \lambda X{<:}T_1.t_2 : \forall X{<:}T_1.T_2} \quad \text{(TA-TABS)}$$

$$\frac{\Gamma \mapsto t_1 : T_1 \qquad \Gamma \mapsto T_1 \Uparrow \forall X{<:}T_{11}.T_{12} \qquad \Gamma \mapsto T_2 <: T_{11}}{\Gamma \mapsto t_1\ [T_2] : [X \mapsto T_2]T_{12}} \quad \text{(TA-TAPP)}$$

▲ 図 28-2.　$F_{<:}$ のアルゴリズム的型付け

これらの規則が全域関数を定義することは容易に確かめられる。さらに、型の露出の結果は、常に変数以外の何かの形を持つ最小の上位型である。例えば、Γ = X<:Top, Y<:Nat→Nat, Z<:Y, W<:Z ならば、

$$\Gamma \mapsto \text{Top} \Uparrow \text{Top} \qquad \Gamma \mapsto \text{Y} \Uparrow \text{Nat}\to\text{Nat} \qquad \Gamma \mapsto \text{W} \Uparrow \text{Nat}\to\text{Nat}$$
$$\Gamma \mapsto \text{X} \Uparrow \text{Top} \qquad \Gamma \mapsto \text{Z} \Uparrow \text{Nat}\to\text{Nat}$$

である。露出の本質的な性質は次のように要約される。

補題 28.1.1.［露出］　Γ ↦ S ⇑ T とするとき、

(1) Γ ⊢ S <: T
(2) Γ ⊢ S <: U かつ U が変数でないならば、Γ ⊢ T <: U　□

［証明］ (1) は Γ ↦ S ⇑ T の導出に関する帰納法による。(2) は Γ ⊢ S <: U の導出に関する帰納法による。　□

28.2　最小型付け

この時点で、最小型を計算するアルゴリズムを、部分型付けを持つ単純型付きラムダ計算の場合と同様にして構築できる。ただし、もう一ひねりとして、関数適用を型検査するときは、図 28-2 に示すように、まず左側の最小型を計算し、次にその型を**露出**して矢印型を得る必要がある。左側の露出が矢印型にならなければ、規則 TA-APP は使えず関数適用は型付けできない。同様に、型適用を型検査するには左側を露出して量化型を得る。

元の型付け規則に対するこのアルゴリズムの健全性と完全性を示すのは容易である。カーネル $F_{<:}$ における証明を与える（フル $F_{<:}$ における議論も同様である。演習 28.2.3 を参照のこと）。

定理 28.2.1.［最小型付け］

(1) Γ ↦ t : T ならば、Γ ⊢ t : T
(2) Γ ⊢ t : T ならば、Γ ↦ t : M かつ Γ ⊢ M <: T　□

［証明］ (1) はアルゴリズム的導出に関する単純な帰納法による。関数適用の場合については補

題 28.1.1 の (1) を用いる。(2) は $\Gamma \vdash \mathtt{t} : \mathtt{T}$ の導出に関する帰納法により、導出の最後に使われた規則について場合分けする。最も興味深いのは T-APP と T-TAPP の場合である。

T-VAR の場合　$\mathtt{t} = \mathtt{x}$　　$\mathtt{x}:\mathtt{T} \in \Gamma$

TA-VAR より、$\Gamma \mapsto \mathtt{x} : \mathtt{T}$。S-REFL より、$\Gamma \vdash \mathtt{T} <: \mathtt{T}$。

T-ABS の場合　$\mathtt{t} = \lambda\mathtt{x}:\mathtt{T}_1.\mathtt{t}_2$　　$\Gamma, \mathtt{x}:\mathtt{T}_1 \vdash \mathtt{t}_2 : \mathtt{T}_2$　　$\mathtt{T} = \mathtt{T}_1 \to \mathtt{T}_2$

帰納法の仮定より、ある $\mathtt{M}_2$ が存在して $\Gamma, \mathtt{x}:\mathtt{T}_1 \mapsto \mathtt{t}_2 : \mathtt{M}_2$ かつ、$\Gamma, \mathtt{x}:\mathtt{T}_1 \vdash \mathtt{M}_2 <: \mathtt{T}_2$（すなわち $\Gamma \vdash \mathtt{M}_2 <: \mathtt{T}_2$。部分型付けは文脈の項の変数束縛に依存しない（補題 26.4.4）ため）である。TA-ABS より、$\Gamma \mapsto \mathtt{t} : \mathtt{T}_1 \to \mathtt{M}_2$。S-REFL と S-ARROW より、$\Gamma \vdash \mathtt{T}_1 \to \mathtt{M}_2 <: \mathtt{T}_1 \to \mathtt{T}_2$。

T-APP の場合　$\mathtt{t} = \mathtt{t}_1\ \mathtt{t}_2$　　$\Gamma \vdash \mathtt{t}_1 : \mathtt{T}_{11} \to \mathtt{T}_{12}$　　$\mathtt{T} = \mathtt{T}_{12}$　　$\Gamma \vdash \mathtt{t}_2 : \mathtt{T}_{11}$

帰納法の仮定より、$\Gamma \mapsto \mathtt{t}_1 : \mathtt{M}_1$ かつ $\Gamma \mapsto \mathtt{t}_2 : \mathtt{M}_2$ が得られ、ここで $\Gamma \vdash \mathtt{M}_1 <: \mathtt{T}_{11} \to \mathtt{T}_{12}$ かつ $\Gamma \vdash \mathtt{M}_2 <: \mathtt{T}_{11}$ である。$\mathtt{N}_1$ を $\mathtt{M}_1$ の最小の非変数の上位型とする。つまり、$\Gamma \mapsto \mathtt{M}_1 \Uparrow \mathtt{N}_1$ である。補題 28.1.1 (2) より、$\Gamma \vdash \mathtt{N}_1 <: \mathtt{T}_{11} \to \mathtt{T}_{12}$。$\mathtt{N}_1$ は変数ではないため、部分型関係の逆転補題（補題 26.4.10）より $\mathtt{N}_1 = \mathtt{N}_{11} \to \mathtt{N}_{12}$ が得られ、ここで $\Gamma \vdash \mathtt{T}_{11} <: \mathtt{N}_{11}$ かつ $\Gamma \vdash \mathtt{N}_{12} <: \mathtt{T}_{12}$ である。推移律により、$\Gamma \vdash \mathtt{M}_2 <: \mathtt{N}_{11}$ であり、規則 TA-APP を適用すれば $\Gamma \mapsto \mathtt{t}_1\ \mathtt{t}_2 : \mathtt{N}_{12}$ が得られ、要求が満たされる。

T-TABS の場合　$\mathtt{t} = \lambda\mathtt{X}<:\mathtt{T}_1.\mathtt{t}_2$　　$\Gamma, \mathtt{X}<:\mathtt{T}_1 \vdash \mathtt{t}_2 : \mathtt{T}_2$　　$\mathtt{T} = \forall\mathtt{X}<:\mathtt{T}_1.\mathtt{T}_2$

帰納法の仮定より、ある $\mathtt{M}_2$ について $\Gamma, \mathtt{X}<:\mathtt{T}_1 \mapsto \mathtt{t}_2 : \mathtt{M}_2$ であり、ここで $\Gamma, \mathtt{X}<:\mathtt{T}_1 \vdash \mathtt{M}_2 <: \mathtt{T}_2$ である。TA-TABS より、$\Gamma \mapsto \mathtt{t} : \forall\mathtt{X}<:\mathtt{T}_1.\mathtt{M}_2$。S-ALL より、$\Gamma \vdash \forall\mathtt{X}<:\mathtt{T}_1.\mathtt{M}_2 <: \forall\mathtt{X}<:\mathtt{T}_1.\mathtt{T}_2$。

T-TAPP の場合　$\mathtt{t} = \mathtt{t}_1\ [\mathtt{T}_2]$　　$\Gamma \vdash \mathtt{t}_1 : \forall\mathtt{X}<:\mathtt{T}_{11}.\mathtt{T}_{12}$

$\mathtt{T} = [\mathtt{X} \mapsto \mathtt{T}_2]\mathtt{T}_{12}$　$\Gamma \vdash \mathtt{T}_2 <: \mathtt{T}_{11}$

帰納法の仮定より、$\Gamma \mapsto \mathtt{t}_1 : \mathtt{M}_1$ が得られ、ここで $\Gamma \vdash \mathtt{M}_1 <: \forall\mathtt{X}<:\mathtt{T}_{11}.\mathtt{T}_{12}$ である。$\mathtt{N}_1$ を $\mathtt{M}_1$ の最小の非変数の上位型とする。つまり、$\Gamma \mapsto \mathtt{M}_1 \Uparrow \mathtt{N}_1$ である。露出補題（補題 28.1.1）より、$\Gamma \vdash \mathtt{N}_1 <: \forall\mathtt{X}<:\mathtt{T}_{11}.\mathtt{T}_{12}$。しかし $\mathtt{N}_1$ は変数でないため、部分型関係の逆転補題（補題 26.4.10）より、$\mathtt{N}_1 = \forall\mathtt{X}<:\mathtt{T}_{11}.\mathtt{N}_{12}$ であり、ここで $\Gamma, \mathtt{X}<:\mathtt{T}_{11} \vdash \mathtt{N}_{12} <: \mathtt{T}_{12}$ である。規則 TA-TAPP より $\Gamma \mapsto \mathtt{t}_1\ [\mathtt{T}_2] : [\mathtt{X} \mapsto \mathtt{T}_2]\mathtt{N}_{12}$ が得られ、代入における部分型付けの保存（補題 26.4.8）より $\Gamma \vdash [\mathtt{X} \mapsto \mathtt{T}_2]\mathtt{N}_{12} <: [\mathtt{X} \mapsto \mathtt{T}_2]\mathtt{T}_{12} = \mathtt{T}$ が得られる。

T-SUB の場合　$\Gamma \vdash \mathtt{t} : \mathtt{S}$　　$\Gamma \vdash \mathtt{S} <: \mathtt{T}$

帰納法の仮定より、$\Gamma \vdash \mathtt{M} <: \mathtt{S}$ の下で $\Gamma \mapsto \mathtt{t} : \mathtt{M}$ である。推移律より $\Gamma \vdash \mathtt{M} <: \mathtt{T}$。 □

系 28.2.2.［型付けの決定可能性］ 部分型関係の決定手続きが存在すれば、カーネル $F_{<:}$ の型付け関係は決定可能である。 □

［証明］ 任意の Γ と $\mathtt{t}$ について、アルゴリズム的型付け規則を用いて $\Gamma \mapsto \mathtt{t} : \mathtt{T}$ の証明を生成することで、$\Gamma \vdash \mathtt{t} : \mathtt{T}$ となる $\mathtt{T}$ の有無を確認できる。もし成功したならば、定理 28.2.1 (1) より $\mathtt{T}$ は元の型付け関係においても $\mathtt{t}$ の型である。さもなくば、定理 28.2.1 (2) より $\mathtt{t}$ は元の型付け関係において型を持たない。特に、アルゴリズム的型付け規則は停止するアルゴリズムに対応することに注意されたい。なぜならこれらの規則は構文主導（与えられた項 $\mathtt{t}$ について高々一つの規則が適用される）であり、下から上へ読むと $\mathtt{t}$ のサイズが常に減少しているためである。 □

演習 28.2.3.［★★］ フル $F_{<:}$ では、上の証明のどこを変えなければならないだろうか。 □

28.3 カーネル $F_{<:}$ の部分型付け

16.1 節において、部分型付けを伴う単純型付きラムダ計算における宣言的な部分型関係は構文主導ではないことに言及した。つまり、部分型付けアルゴリズムとして直接には解釈できないのである。これは二つの理由による。(1) S-REFL と S-TRANS の結論は他の規則と重複する（このため規則を下から上に読んでも、どの規則の適用を試みるべきかわからない）こと、そして (2) S-TRANS の前提が、結論には現れないメタ変数（素朴なアルゴリズムだと何らかの方法で「推測」しなければならない）に言及していることである。これらの問題は障害となる二つの規則を型システムから取り除けば修正できることを示したが、その前に、レコードの部分型付けの三つの規則を一つに結合することで、型システムを少しばかり修正しなければならなかった。

カーネル $F_{<:}$ についても、話は同様である。やはり障害となる規則は S-REFL と S-TRANS であり、これらの規則を取り除き、取り除いた規則を実質的に使っていた部分を埋めるよう残りの規則を少し修正すれば、アルゴリズムが得られる。

部分型付けを伴う単純型付きラムダ計算においては、反射律の規則は実質的に使われていなかった。このため、導出可能な部分型付け判断式を変えることなく、単に取り除くことができた（補題 16.1.2 (1)）。一方 $F_{<:}$ では、$\Gamma \vdash X <: X$ の形の部分型付け判断式は反射律でのみ証明できる。このため、反射律の規則を全く取り除いてしまう代わりに、変数にのみ適用される、制限された反射律の公理を追加すべきである。

$$\Gamma \vdash X <: X$$

同様に、S-TRANS を除去するには、まず最初にどの部分で実質的に使われているのかを理解しなければならない。ここに、S-TVAR 規則との興味深い相互作用がある。S-TVAR は型変数に関する仮定を部分型付け判断式の導出に使えるようにする。例えば、もし $\Gamma = W{<:}Top, X{<:}W, Y{<:}X, Z{<:}Y$ ならば、部分型付け判断式 $\Gamma \vdash Z <: W$ は S-TRANS が取り除かれた型システムでは証明できない。左側の部分導出が S-TVAR であるような、次のような S-TRANS のインスタンス

$$\dfrac{\dfrac{Z <: Y \in \Gamma}{\Gamma \vdash Z <: Y}\ (\text{S-TVAR}) \qquad \dfrac{\vdots}{\Gamma \vdash Y <: W}}{\Gamma \vdash Z <: W}\ (\text{S-TRANS})$$

は、一般に除去できない。

幸いなことに、部分型付けにおいて推移律を本質的に使わなければならないのはこの形の導出**だけ**である。この観測を正確にするには、型変数の探索の規則を用いた直後に推移律を用いるパターンを捉える新しい部分型規則

$$\dfrac{X{<:}U \in \Gamma \qquad \Gamma \vdash U <: T}{\Gamma \vdash X <: T}$$

を導入し、さらに推移律と変数の規則をこの規則で置き換えても導出可能な部分型付け判断式の集合は変わらないことを示せばよい。

この変更により、カーネル $F_{<:}$ のアルゴリズム的部分型関係が得られる。これを図 28-3 に示す。アルゴリズム的部分型付け判断式と元の部分型付け判断式の両方がかかわる議論において両者を区別できるよう、アルゴリズム的部分型付け判断式のほうではターンスタイルの記号に矢印を加えた。

$\rightarrow \forall$ <: Top　　　　$\lambda_{<:}$（図 16-2）の拡張

アルゴリズム的部分型付け　　$\boxed{\Gamma \mapsto S <: T}$

$$\Gamma \mapsto S <: \mathsf{Top} \qquad \text{(SA-TOP)}$$

$$\Gamma \mapsto X <: X \qquad \text{(SA-REFL-TVAR)}$$

$$\frac{X{<:}U \in \Gamma \qquad \Gamma \mapsto U <: T}{\Gamma \mapsto X <: T} \qquad \text{(SA-TRANS-TVAR)}$$

$$\frac{\Gamma \mapsto T_1 <: S_1 \qquad \Gamma \mapsto S_2 <: T_2}{\Gamma \mapsto S_1 \rightarrow S_2 <: T_1 \rightarrow T_2} \qquad \text{(SA-ARROW)}$$

$$\frac{\Gamma, X{<:}U_1 \mapsto S_2 <: T_2}{\Gamma \mapsto \forall X{<:}U_1.S_2 <: \forall X{<:}U_1.T_2} \qquad \text{(SA-ALL)}$$

▲ 図 28-3.　カーネル $F_{<:}$ のアルゴリズム的部分型付け

新しい SA-REFL-TVAR と SA-TRANS-TVAR の規則が古い反射律と推移律の規則の置き換えとして十分であるという事実は次の二つの補題で捉えられる。

補題 28.3.1.［アルゴリズム的部分型関係の反射律］ $\Gamma \mapsto T <: T$ はすべての Γ と T について証明できる。 □

［証明］ T に関する帰納法による。 □

補題 28.3.2.［アルゴリズム的部分型付けの推移律］ $\Gamma \mapsto S <: Q$ かつ $\Gamma \mapsto Q <: T$ ならば $\Gamma \mapsto S <: T$ □

［証明］ 二つの導出のサイズの和に関する帰納法による。二つの導出を仮定し、両方の最後で使われた規則の場合分けによって進める。

もし右側の導出が SA-TOP のインスタンスならば、SA-TOP より $\Gamma \vdash S <: \mathsf{Top}$ であるため証明完了である。もし左側の導出が SA-TOP のインスタンスならば、$Q = \mathsf{Top}$ であり、アルゴリズム的規則を見れば右側の導出もまた SA-TOP のインスタンスでなければならないことがわかる。

もし一方の導出が SA-REFL-TVAR の導出ならば、他方が求めていた結果であるため、やはり証明完了である。

もし左側の導出が SA-TRANS-TVAR のインスタンスで終了しているならば、$Y{<:}U \in \Gamma$ の下で $S = Y$ であり結論 $\Gamma \vdash U <: Q$ を持つ部分導出がある。帰納法の仮定より $\Gamma \mapsto U <: T$ であり、SA-TRANS-TVAR より、所要の $\Gamma \mapsto Y <: T$ が得られる。

もし左側の導出が S-ARROW のインスタンスで終了しているならば、$S = S_1 \rightarrow S_2$ かつ $Q = Q_1 \rightarrow Q_2$ でありここで部分導出において $\Gamma \mapsto Q_1 <: S_1$ かつ $\Gamma \mapsto S_2 <: Q_2$ である。しかし、右側の導出が SA-TOP の場合は考慮済みであるので、残った可能性は導出が SA-ARROW で終わる場合のみである。それゆえ $T = T_1 \rightarrow T_2$ とさらなる二つの部分導出 $\Gamma \mapsto T_1 <: Q_1$ と $\Gamma \mapsto Q_2 <: T_2$ が得られる。ここで帰納法の仮定を二度適用すれば $\Gamma \mapsto T_1 <: S_1$ と $\Gamma \mapsto S_2 <: T_2$ が得られる。SA-ARROW より、所要の $\Gamma \mapsto S_1 \rightarrow S_2 <: T_1 \rightarrow T_2$ が得られる。

もし左側の導出が SA-ALL のインスタンスで終了している場合、同様の議論が成り立つ。$S = \forall X{<:}U_1.S_2$ と $Q = \forall X{<:}U_1.Q_2$ が得られ、部分導出は $\Gamma, X{<:}U_1 \mapsto S_2 <: Q_2$ である。しかしさらに、右側の導出が SA-TOP である場合は考慮済みであるため、SA-ALL で終了しているはずである。このため、$T = \forall X{<:}U_1.T_2$ であり、部分導出は $\Gamma, X{<:}U_1 \mapsto Q_2 <: T_2$ である。帰納法の仮定を適用すれば $\Gamma, X{<:}U_1 \mapsto S_2 <: T_2$ が得られ、SA-ALL より $\Gamma \mapsto \forall X{<:}U_1.S_2 <: \forall X{<:}U_1.T_2$ が得られる。

□

定理 28.3.3.［アルゴリズム的部分型付けの健全性と完全性］ $\Gamma \vdash S <: T$ のとき、かつそのときのみ $\Gamma \mapsto S <: T$ □

［証明］ どちらの方向も導出に関する帰納法により導く。健全性（⇐）は定型作業である。完全性（⇒）は補題 28.3.1 と補題 28.3.2 に依存する。 □

最後に、部分型付け規則が**全域の**アルゴリズムを定義していること、つまりすべての入力で停止すること、を確かめる必要がある。これは、部分型付け判断式を重み付けし、すべてのアルゴリズム的規則の結論が前提より真に重いことを確かめることにより行う。

定義 28.3.4. 文脈 Γ における型 T の**重み**は $weight_\Gamma(\mathsf{T})$ と書き、次のように定義される[†1]。

$$
\begin{array}{lcll}
weight_\Gamma(\mathsf{X}) & = & weight_{\Gamma_1}(\mathsf{U}) + 1 & \Gamma = \Gamma_1, \mathsf{X}{<:}\mathsf{U}, \Gamma_2 \text{ の場合} \\
weight_\Gamma(\mathsf{Top}) & = & 1 & \\
weight_\Gamma(\mathsf{T}_1 \to \mathsf{T}_2) & = & weight_\Gamma(\mathsf{T}_1) + weight_\Gamma(\mathsf{T}_2) + 1 & \\
weight_\Gamma(\forall \mathsf{X}{<:}\mathsf{T}_1 . \mathsf{T}_2) & = & weight_{\Gamma, \mathsf{X}<:\mathsf{T}_1}(\mathsf{T}_2) + 1 &
\end{array}
$$

部分型付け判断式 $\Gamma \vdash S <: T$ の**重み**は Γ における S と T の重みの総和である。 □

定理 28.3.5. 部分型付けアルゴリズムはすべての入力について停止する。 □

［証明］ アルゴリズム的部分型付け規則のインスタンスにおける結論の重みはどの前提の重みよりも真に大きい。 □

系 28.3.6. カーネル $F_{<:}$ の部分型付けは決定可能である。 □

28.4 フル $F_{<:}$ の部分型付け

図 28-4 に要約したフル $F_{<:}$ の部分型検査アルゴリズムは、カーネル $F_{<:}$ とほとんど同じである。唯一の違いは SA-ALL をより柔軟にしたもので置き換えたことである。カーネル $F_{<:}$ と同様に、このアルゴリズム的関係の、元の部分型関係に対する健全性と完全性は、アルゴリズム的関係が反射的で推移的である事実から直接導かれる。

反射律については、議論は前と全く同じであるが、推移律は少しだけ扱いづらい。その理由は、前節のカーネル $F_{<:}$ の推移律の証明（補題 28.3.2）を思い出せばわかる。そこでのアイディアは、部分型付け判断式 $\Gamma \vdash S <: Q$ と $\Gamma \vdash Q <: T$ で終わる二つの部分型付け導出を取り、（帰納法の仮定より）より小さな導出の組に推移律が使えると仮定し、どうやってこれらの部分導出を再配置して再び組み立て、推移律の規則を使わずに $\Gamma \vdash S <: T$ の導出を構築するかを示すことだった。ここで、新しい SA-ALL 規則で終わる二つの導出があるとしよう。

$$
\dfrac{\dfrac{\vdots}{\Gamma \vdash Q_1 <: S_1} \quad \dfrac{\vdots}{\Gamma, X{<:}Q_1 \vdash S_2 <: Q_2}}{\Gamma \vdash \forall X{<:}S_1 . S_2 <: \forall X{<:}Q_1 . Q_2}
\qquad
\dfrac{\dfrac{\vdots}{\Gamma \vdash T_1 <: Q_1} \quad \dfrac{\vdots}{\Gamma, X{<:}T_1 \vdash Q_2 <: T_2}}{\Gamma \vdash \forall X{<:}Q_1 . Q_2 <: \forall X{<:}T_1 . T_2}
$$

前の証明のパターンを真似て、帰納法の仮定を用いて左右の部分導出を結合し、結論が $\Gamma \vdash \forall X{<:}S_1 . S_2 <: \forall X{<:}T_1 . T_2$ であるような SA-ALL を一回用いるだけで導出を終了したい。それぞれの左側の二つの部分導出については何も問題ない。帰納法の仮定から、推移律を使わずに

[†1] （訳注：原著正誤表からの補足）Γ に含まれるすべての型のサイズの合計と、ここで重みを定義しようとしている型 T のサイズとの和は、再帰呼び出しの際に常に減少するため、この $weight_\Gamma$ の定義は正しい定義になっている。

→ ∀ <: Top フル　　図 28-3 の拡張

アルゴリズム的部分型付け　$\boxed{\Gamma \mapsto S <: T}$

$$\Gamma \mapsto S <: Top \quad \text{(SA-Top)}$$

$$\Gamma \mapsto X <: X \quad \text{(SA-Refl-TVar)}$$

$$\frac{X<:U \in \Gamma \qquad \Gamma \mapsto U <: T}{\Gamma \mapsto X <: T} \quad \text{(SA-Trans-TVar)}$$

$$\frac{\Gamma \mapsto T_1 <: S_1 \qquad \Gamma \mapsto S_2 <: T_2}{\Gamma \mapsto S_1 \to S_2 <: T_1 \to T_2} \quad \text{(SA-Arrow)}$$

$$\frac{\Gamma \mapsto T_1 <: S_1 \qquad \Gamma, X<:T_1 \mapsto S_2 <: T_2}{\Gamma \mapsto \forall X<:S_1.S_2 <: \forall X<:T_1.T_2} \quad \text{(SA-All)}$$

▲ 図 28-4.　フル $F_{<:}$ のアルゴリズム的部分型付け

$\Gamma \vdash T_1 <: S_1$ の導出を得られるためである。しかしながら右側の二つの部分導出には、部分導出における文脈が異なるため帰納法の仮定が使えない。一方では X の上界が Q_1 であり、他方では T_1 である。

幸運なことに、文脈を**同じにする**方法はわかっている。第 26 章（補題 26.4.5）の**狭化**の性質より、妥当な部分型付け判断式は、文脈において上界を部分型に置き換えても妥当性が保たれることがわかる。このため、部分導出 $\Gamma, X<:Q_1 \vdash S_2 <: Q_2$ は $\Gamma, X<:T_1 \vdash S_2 <: Q_2$ へと狭化でき、これによって帰納法の仮定が使えるようになるように見える。

しかしながら、少し注意深くあらねばならない。補題 26.4.5 よりどんな導出からも狭化された結論を持つ導出を構築できるが、それは新しい導出が元のものと同じ大きさであることを保証**しない**。実際、この補題の証明を調べると明らかに、狭化は一般的にはより大きな導出を生んでしまう。なぜなら狭化される変数を探索するために公理 S-TVar が用いられるところに、任意に大きな導出のコピーの結合が発生するためである。さらにこの結合操作は、今まさにこの型システムで許容されることを示そうとしている推移律の新しいインスタンスを作ってしまうのである。

この困難に対する解決策は推移律と狭化を**同時に**証明することである。ここで推移律における中間の型 Q と、狭化における元の上界 Q のサイズに関する帰納法の仮定を用いる。

この議論を始めるにあたり、まずは簡単な補題を示す。文脈を並べ替えたり、文脈にフレッシュな型変数の束縛を追加したりしても、導出可能だった部分型付け判断式はどれも導出不可能にはならないということを示すものである。

補題 28.4.1. ［並べ替えと弱化］

(1) Δ を Γ の正しい形式の並べ替えとする（補題 26.4.1 を参照）。$\Gamma \mapsto S <: T$ ならば、$\Delta \mapsto S <: T$

(2) $\Gamma \mapsto S <: T$ かつ $dom(\Delta) \cap dom(\Gamma) = \emptyset$ ならば $\Gamma, \Delta \mapsto S <: T$ □

［証明］ 定型的な帰納法による。(1) は (2) の SA-All の場合に用いられる。 □

補題 28.4.2. ［フル $F_{<:}$ における推移律と狭化］

(1) $\Gamma \mapsto S <: Q$ かつ $\Gamma \mapsto Q <: T$ ならば、$\Gamma \mapsto S <: T$

(2) $\Gamma, X<:Q, \Delta \mapsto M <: N$ かつ $\Gamma \mapsto P <: Q$ ならば $\Gamma, X<:P, \Delta \mapsto M <: N$ □

［証明］ 二つの部分を Q のサイズに関する帰納法で同時に証明する。帰納法の各段階において、(2) の議論では対象となる Q について (1) が既に成立しているものとする。(1) は真に小さい Q についてのみ (2) を用いる。

(1) 一つ目の導出のサイズに関する帰納法を内側で回し、両方の導出の最後の規則に関して場合分けする。すべての場合は、一つを除いて補題 28.3.2 と同様である。違いは SA-ALL の場合に生じる。

右側の導出が SA-TOP のインスタンスならば、SA-TOP より $\Gamma \vdash \mathsf{S} <: \mathsf{Top}$ であるため完了である。左側の導出が SA-TOP のインスタンスならば $\mathsf{Q} = \mathsf{Top}$ であり、かつアルゴリズム的規則を調べると、右側の導出もまた SA-TOP でなければならないことがわかる。もしどちらか一方の導出が SA-REFL-TVAR のインスタンスならば、他方の導出が求めていた結果であるため完了である。

もし左側の導出が SA-TRANS-TVAR のインスタンスで終了しているならば、$\mathsf{Y}<:\mathsf{U} \in \Gamma$ なる Y が存在し S=Y である。ここで部分導出は $\Gamma \vdash \mathsf{U} <: \mathsf{Q}$ である。内側の帰納法の仮定より $\Gamma \mapsto \mathsf{U} <: \mathsf{T}$ であり、再び SA-TRANS-TVAR より所要の $\Gamma \mapsto \mathsf{Y} <: \mathsf{T}$ が得られる。

もし左側の導出が SA-ARROW か SA-ALL のインスタンスで終了しているならば、右側については SA-TOP で終了する場合は考慮済みであるため、左側と同じ規則で終了しているはずである。もしこの規則が SA-ARROW ならば、$\mathsf{S} = \mathsf{S}_1 \rightarrow \mathsf{S}_2$ かつ $\mathsf{Q} = \mathsf{Q}_1 \rightarrow \mathsf{Q}_2$ かつ $\mathsf{T} = \mathsf{T}_1 \rightarrow \mathsf{T}_2$ であり、部分導出は $\Gamma \mapsto \mathsf{Q}_1 <: \mathsf{S}_1$、$\Gamma \mapsto \mathsf{S}_2 <: \mathsf{Q}_2$、$\Gamma \mapsto \mathsf{T}_1 <: \mathsf{Q}_1$、$\Gamma \mapsto \mathsf{Q}_2 <: \mathsf{T}_2$ である。外側の帰納法の仮定の (1) を二度使えば（Q_1 と Q_2 は両方とも Q より小さいことに注意）、$\Gamma \mapsto \mathsf{T}_1 <: \mathsf{S}_1$ と $\Gamma \mapsto \mathsf{S}_2 <: \mathsf{T}_2$ が得られ、SA-ARROW より $\Gamma \mapsto \mathsf{S}_1 \rightarrow \mathsf{S}_2 <: \mathsf{T}_1 \rightarrow \mathsf{T}_2$ である。

二つの導出が SA-ALL で終了しているとき、$\mathsf{S} = \forall \mathsf{X}<:\mathsf{S}_1.\mathsf{S}_2$ かつ $\mathsf{Q} = \forall \mathsf{X}<:\mathsf{Q}_1.\mathsf{Q}_2$ かつ $\mathsf{T} = \forall \mathsf{X}<:\mathsf{T}_1.\mathsf{T}_2$ である。ただし部分導出は

$$\begin{array}{ll} \Gamma \mapsto \mathsf{Q}_1 <: \mathsf{S}_1 & \Gamma, \mathsf{X}<:\mathsf{Q}_1 \mapsto \mathsf{S}_2 <: \mathsf{Q}_2 \\ \Gamma \mapsto \mathsf{T}_1 <: \mathsf{Q}_1 & \Gamma, \mathsf{X}<:\mathsf{T}_1 \mapsto \mathsf{Q}_2 <: \mathsf{T}_2 \end{array}$$

である。外側の帰納法の仮定の (1)（Q_1 は Q より小さい）より、上界に関する二つの部分導出を結合し $\Gamma \mapsto \mathsf{T}_1 <: \mathsf{S}_1$ を得る。本体については、文脈が少し異なるため、もう少し頑張る必要がある。最初に外側の帰納法の仮定の (2) を使い（再度、Q_1 は Q より小さいことに注意）、$\Gamma, \mathsf{X}<:\mathsf{Q}_1 \mapsto \mathsf{S}_2 <: \mathsf{Q}_2$ の導出における上界 X を狭化することで、$\Gamma, \mathsf{X}<:\mathsf{T}_1 \mapsto \mathsf{S}_2 <: \mathsf{Q}_2$ が得られる。ここで外側の帰納法の (1) が適用でき（Q_2 は Q より小さい）、$\Gamma, \mathsf{X}<:\mathsf{T}_1 \mapsto \mathsf{S}_2 <: \mathsf{T}_2$ が得られる。最後に、SA-ALL より $\Gamma \mapsto \forall \mathsf{X}<:\mathsf{S}_1.\mathsf{S}_2 <: \forall \mathsf{X}<:\mathsf{T}_1.\mathsf{T}_2$。

(2) 再び、一つ目の導出の大きさに関する帰納法を内側で回し、この導出の最後の規則に関して場合分けする。ほとんどの場合は内側の帰納法の仮定を単純に用いて進める。興味深いのは $\mathsf{M} = \mathsf{X}$ のときの SA-TRANS-TVAR の場合である。部分導出は $\Gamma, \mathsf{X}<:\mathsf{Q}, \Delta \mapsto \mathsf{Q} <: \mathsf{N}$ である。内側の帰納法の仮定をこの部分導出に適用すれば $\Gamma, \mathsf{X}<:\mathsf{P}, \Delta \mapsto \mathsf{Q} <: \mathsf{N}$ が得られる。さらに、弱化（補題 28.4.1 (2)）を二つ目の導出に適用すれば $\Gamma, \mathsf{X}<:\mathsf{P}, \Delta \mapsto \mathsf{P} <: \mathsf{Q}$ が得られる。ここで、外側の帰納法の仮定の (1)（同じ Q に関して）より、$\Gamma, \mathsf{X}<:\mathsf{P}, \Delta \mapsto \mathsf{P} <: \mathsf{N}$ が得られる。規則 SA-TRANS-TVAR より、所要の $\Gamma, \mathsf{X}<:\mathsf{P}, \Delta \mapsto \mathsf{X} <: \mathsf{N}$ が得られる。 □

演習 28.4.3. [★★★★ ↛] 量化子の部分型付け規則として合理的なものには他に、カーネル $F_{<:}$ よりも少しばかり柔軟だが、フル $F_{<:}$ よりも本質的に柔軟でないものがある。

$$\frac{\Gamma \vdash \mathsf{S}_1 <: \mathsf{T}_1 \qquad \Gamma \vdash \mathsf{T}_1 <: \mathsf{S}_1 \qquad \Gamma, \mathsf{X}<:\mathsf{T}_1 \vdash \mathsf{S}_2 <: \mathsf{T}_2}{\Gamma \vdash \forall \mathsf{X}<:\mathsf{S}_1.\mathsf{S}_2 <: \forall \mathsf{X}<:\mathsf{T}_1.\mathsf{T}_2} \qquad \text{(S-ALL)}$$

この規則はカーネル $F_{<:}$ に近いものの、二つの上界が構文的に同じであることを要求する代わりに、同値な型、つまりお互いに部分型になっていることだけを要求する。カーネルの規則との違いは、非自明な同値類を生成する部分型付け規則を持つ何らかの機能（例えばレコード）で言語を拡張したときのみに現れる。例えば、レコード付きの純粋なカーネル $F_{<:}$ では、型 $\forall \mathsf{X}<:\{\mathsf{a}:\mathsf{Top},\mathsf{b}:\mathsf{Top}\}.\mathsf{X}$

は ∀X<:{b:Top,a:Top}.X の部分型とはならないが、上の規則によれば部分型になる。この規則を持つ型システムに対する部分型付けは決定可能だろうか？ □

28.5 フル $F_{<:}$ の決定不能性

前節でフル $F_{<:}$ のアルゴリズム的部分型付け規則は健全かつ完全であることを示した。すなわち、これらの規則に閉じた最小の関係は元の宣言的な規則に閉じた最小の関係と同じ部分型付け判断式を含むのである。ここで、これらの規則を実装するアルゴリズムはすべての入力で停止するのかという疑問が残る。残念ながら、そして発見当時の人々も大変驚いたのだが、そうではない。

演習 28.5.1. [★] もしフル $F_{<:}$ のアルゴリズム的規則で定義されるアルゴリズムが常に停止するわけではないならば、カーネル $F_{<:}$ のアルゴリズムの停止性の証明をフルの型システムの規則に持ってくることは明らかに不可能である。正確にはどこで破綻するだろうか。 □

Ghelli [173] による、部分型付けアルゴリズムを発散させる例がある。まず次の略記を定義する。

$$\neg S \stackrel{\text{def}}{=} \forall X<:S.X$$

¬ 演算子の重要な性質は、部分型付け判断式の右辺と左辺を入れ替えられることである。

事実 28.5.2. $\Gamma \vdash \neg S <: \neg T$ のとき、かつそのときのみ $\Gamma \vdash T <: S$ □

［証明］ 演習［★★ ↛］。 □

さて、型 T を次のように定義しよう。

$$T = \forall X<:Top.\ \neg(\forall Y<:X.\neg Y)$$

もしアルゴリズム的部分型付けを下から上に用いて、次の部分型付け判断式

$$X_0<:T \;\mapsto\; X_0 \;<:\; \forall X_1<:X_0.\neg X_1$$

の導出を構築しようとすると、どんどん大きくなる部分ゴールの無限後退に陥ってしまう。

$$\begin{array}{llll}
X_0<:T & \mapsto & X_0 & <: \ \forall X_1<:X_0.\neg X_1 \\
X_0<:T & \mapsto & \forall X_1<:Top.\ \neg(\forall X_2<:X_1.\neg X_2) & <: \ \forall X_1<:X_0.\neg X_1 \\
X_0<:T, X_1<:X_0 & \mapsto & \neg(\forall X_2<:X_1.\neg X_2) & <: \ \neg X_1 \\
X_0<:T, X_1<:X_0 & \mapsto & X_1 & <: \ \forall X_2<:X_1.\neg X_2 \\
X_0<:T, X_1<:X_0 & \mapsto & X_0 & <: \ \forall X_2<:X_1.\neg X_2
\end{array}$$

(以下略)

新しい変数が文脈に追加される際、文脈が正しい形式であることを保つべく、暗黙に名前を付け替える。ここでは無限後退のパターンを明確にするよう新しい名前を選んでいる。重要なトリックは「上界の再設定」である。これは、例えば、二行目と三行目の間で起きている。左辺の X_1 の上界は、二行目では Top だったのが、三行目では X_0 に変わっている。二行目の左辺全体はそれ自体が X_0 の上界であるため、この上界の再設定は各ループにおいて走査せねばならない変数の列が文脈中でどんどん長くなる循環のパターンを作っている。(読者はこの例の背景となる**意味論的**な直観を探したりしないよう注意せよ。特に、¬T は単なる構文的な意味での否定である。)

さらに悪いことに、この特定のアルゴリズムが幾つかの入力で停止しなくなるだけでなく、元のフル $F_{<:}$ の部分型関係に対して健全かつ完全ですべての入力において停止するようなアルゴリズムが**存在しない**ことが示せてしまう（Pierce [366]）。この事実の証明は本書で扱うには大きすぎる。しかしながら、その雰囲気をつかむため、もう一つの例を見てみよう。

定義 28.5.3. 型 T における**正**と**負**の出現を次のように定義する。T 自身は T の正の出現である。$T_1 \to T_2$ が正（負）の出現ならば、T_1 は負（正）の出現であり、T_2 は正（負）の出現である。$\forall X<:T_1.T_2$ が正（負）の出現ならば、T_1 は負（正）の出現であり、T_2 は正（負）の出現である。部分型付け判断式 $\Gamma \vdash S <: T$ における正と負の出現を次のように定義する。型 S と、Γ の中の型変数の上界は負の出現である。型 T は正の出現である。 □

「正」と「負」という言葉は論理学に由来する。命題と型の間のよく知られた **Curry-Howard 対応**（9.4 節）によれば、型 $S \to T$ は論理的な命題 $S \Rightarrow T$ に対応し、これは論理的含意の定義より（古典論理においては）$\neg S \vee T$ と同値である。ここで明らかに、部分命題 S が「負」の位置（つまり含意全体が奇数個の否定の中）にあるとき、かつそのときのみ、含意全体は偶数個の否定の中にある。T における正の出現は $\neg T$ における負の出現に対応することに注意せよ。

事実 28.5.4. X が S の正の位置でのみ、かつ T の負の位置でのみ出現するとする。このとき $X<:U \vdash S <: T$ と $\vdash [X \mapsto U]S <: [X \mapsto U]T$ は同値である。 □

［証明］ 演習［★★ ↛］。 □

さて、T を次の型とし、

$$
\begin{aligned}
T \quad = \quad & \forall X_0<:Top.\, \forall X_1<:Top.\, \forall X_2<:Top. \\
& \quad \neg(\forall Y_0<:X_0.\, \forall Y_1<:X_1.\, \forall Y_2<:X_2.\, \neg X_0)
\end{aligned}
$$

そして次の部分型付け判断式を考える。

$$
\begin{aligned}
\vdash \quad T \quad <: \quad & \forall X_0<:T.\, \forall X_1<:P.\, \forall X_2<:Q. \\
& \quad \neg(\forall Y_0<:Top.\, \forall Y_1<:Top.\, \forall Y_2<:Top. \\
& \quad\quad \neg(\forall Z_0<:Y_0.\, \forall Z_1<:Y_2.\, \forall Z_2<:Y_1.\, U))
\end{aligned}
$$

この判断式は、ある単純な計算機の状態の記述とみなせる。変数 X_1 と X_2 はこの機械の「レジスタ」である。現在の内容は型 P と Q である。この機械の「命令列」は三行目である。最初の命令は変数 Z_1 と Z_2 の上界（Y_2 と Y_1、順序に注意）に埋め込まれており、未指定の型 U はプログラムの残りの命令である。型 T、ネストした否定と束縛変数 X_0 と Y_0 はより単純な上の例と同じ役割を果たす。つまりこれらにより「クランクを回す」ことができ、元のゴールと同じ形をした部分ゴールに戻れるようになる。クランクの一回転が機械の一サイクルに対応するだろう。

この例では、命令列の先頭の命令は、「レジスタ 1 と 2 の中身を交換せよ」という指令を表現している。これを確認するために、上の二つの事実を用いて次のように計算する（追跡しやすいように、二つのレジスタの中の値 P と Q に網掛けを施してある）。

$$
\begin{array}{lll}
& \vdash & \mathsf{T} \\
& <: & \forall X_0 <: \mathsf{T}.\ \forall X_1 <: \mathsf{P}.\ \forall X_2 <: \mathsf{Q}. \\
& & \quad \neg(\forall Y_0 <: \mathsf{Top}.\ \forall Y_1 <: \mathsf{Top}.\ \forall Y_2 <: \mathsf{Top}. \\
& & \qquad \neg(\forall Z_0 <: Y_0.\ \forall Z_1 <: Y_2.\ \forall Z_2 <: Y_1.\ \mathsf{U})) \\
\Longleftrightarrow & \vdash & \neg(\forall Y_0 <: \mathsf{T}.\ \forall Y_1 <: \mathsf{P}.\ \forall Y_2 <: \mathsf{Q}.\ \neg\mathsf{T}) \\
& <: & \neg(\forall Y_0 <: \mathsf{Top}.\ \forall Y_1 <: \mathsf{Top}.\ \forall Y_2 <: \mathsf{Top}. \qquad \text{事実 28.5.4 による} \\
& & \quad \neg(\forall Z_0 <: Y_0.\ \forall Z_1 <: Y_2.\ \forall Z_2 <: Y_1.\ \mathsf{U})) \\
\Longleftrightarrow & \vdash & (\forall Y_0 <: \mathsf{Top}.\ \forall Y_1 <: \mathsf{Top}.\ \forall Y_2 <: \mathsf{Top}. \\
& & \quad \neg(\forall Z_0 <: Y_0.\ \forall Z_1 <: Y_2.\ \forall Z_2 <: Y_1.\ \mathsf{U})) \\
& <: & (\forall Y_0 <: \mathsf{T}.\ \forall Y_1 <: \mathsf{P}.\ \forall Y_2 <: \mathsf{Q}.\ \neg\mathsf{T}) \qquad \text{事実 28.5.2 による} \\
\Longleftrightarrow & \vdash & \neg(\forall Z_0 <: \mathsf{T}.\ \forall Z_1 <: \mathsf{Q}.\ \forall Z_2 <: \mathsf{P}.\ \mathsf{U})) \\
& <: & \neg\mathsf{T} \qquad \text{事実 28.5.4 による} \\
\Longleftrightarrow & \vdash & \mathsf{T} \\
& <: & (\forall Z_0 <: \mathsf{T}.\ \forall Z_1 <: \mathsf{Q}.\ \forall Z_2 <: \mathsf{P}.\ \mathsf{U})) \qquad \text{事実 28.5.2 による}
\end{array}
$$

注意したいのは、導出の最後において、値 P と Q の位置が切り替わっただけでなく、それを引き起こした命令が処理中に消費されていることである。命令列の先頭には、次に「実行される」ものとして U が残される。ちょうど今実行した命令と同じように始まる U を選ぶことで、

$$
\begin{array}{lll}
\mathsf{U} & = & \neg(\forall Y_0 <: \mathsf{Top}.\ \forall Y_1 <: \mathsf{Top}.\ \forall Y_2 <: \mathsf{Top}. \\
& & \quad \neg(\forall Z_0 <: Y_0.\ \forall Z_1 <: Y_2.\ \forall Z_2 <: Y_1.\ \mathsf{U}'))
\end{array}
$$

もう一度入れ替えを実行し、U′ に継続する前にレジスタを元の状態に戻せる。他にも、異なる種類の振る舞いが起こるように U の値を選ぶこともできる。例えば、

$$
\begin{array}{lll}
\mathsf{U} & = & \neg(\forall Y_0 <: \mathsf{Top}.\ \forall Y_1 <: \mathsf{Top}.\ \forall Y_2 <: \mathsf{Top}. \\
& & \quad \neg(\forall Z_0 <: Y_0.\ \forall Z_1 <: Y_1.\ \forall Z_2 <: Y_2.\ Y_1))
\end{array}
$$

ならば、機械の次のサイクルにおいて、レジスタ 1 の現在の値（すなわち Q）が U の位置に現れる。つまり実効的には、レジスタ 1 を介し、Q で表される命令列へ「間接分岐」する。分岐構文と算術演算（後者関数、前者関数、ゼロ判定）はこのトリックの一般化を用いることで表現できる。

まとめると、2 カウンタ機械から部分型付け判断式へ帰着させることで決定性の証明に辿り着く。2 カウンタ機械とは通常のチューリング機械の単純な変種であり、有限の制御命令と、自然数を保持する二つのカウンタからなるものである。

定理 28.5.5.［Pierce, 1994 [366]］ 2 カウンタ機械 M について、部分型付け判断式 $S(M)$ が存在し、$S(M)$ がフル $F_{<:}$ において導出可能であるとき、かつそのときのみ、M の実行が停止する。 □

このことから、任意の部分型付け判断式が証明可能かどうか決定できるとしたら、任意の 2 カウンタ機械が最終的に停止するかどうかを決定できてしまうだろう。2 カウンタ機械の停止性は決定不能である（Hopcroft and Ullman [218] 参照）ので、フル $F_{<:}$ の部分型付け問題も決定不能である。

ここで再び強調したいのは、部分型関係の決定不能性は 28.4 節で説明した部分型付けの半アルゴリズムが健全でないことや完全でないことを意味しないということである。もし部分型付け判断式 $\Gamma \vdash \mathsf{S} <: \mathsf{T}$ が宣言的な部分型付け規則で証明可能ならば、アルゴリズムは必ず停止し *true* を返す。もし $\Gamma \vdash \mathsf{S} <: \mathsf{T}$ が宣言的な部分型付け規則で証明可能**でない**ならば、アルゴリズムは発散するか *false* を返す。問題は、アルゴリズム的規則においては、部分型付け判断式の証明が失敗する方法が

二通りあるということである。部分ゴールの無限の列を生成する（その結論を持つ有限の導出が存在しないことを意味する）か、Top <: S→T のような明らかな不整合に達することである。部分型付けのアルゴリズムは一方は検出できるが、他方はできない。

フル $F_{<:}$ の決定不能性は型システムが実用に値しないことを意味するのだろうか？ 実際には、$F_{<:}$ の決定不能性は、**それ自体では**ひどく深刻な欠陥ではないと一般に考えられている。一つには、部分型検査器を発散させるには三つのかなり特別な性質を持つゴールを与える必要があることが示されており（Ghelli [173]）、それぞれの性質はプログラマが偶然作ってしまうとは想像しづらい。また、広く用いられている言語の中にも、型検査や型再構築問題が原理的には極めて遅いもの（例えば ML や Haskell。262 ページで見た）や、決定可能ですらないもの（例えば C++ や λProlog（Felty, Gunter, Hannan, Miller, Nadathur, and Scedrov [147]））がある。実際には、決定不能性よりも、次節で述べる結びと交わりの欠落（演習 28.6.3 参照）のほうが、フル $F_{<:}$ の非常に重大な欠点であると経験的に知られている。

演習 28.5.6. [★★★★]　(1) フル $F_{<:}$ の変種で、Top 型がなく変数の束縛として X<:T と X の両方がある（つまり、有界量化と非有界量化の両方）ものを定義せよ。この変種は**完全有界量化**と呼ばれている。(2) この型システムについて部分型関係が決定可能であることを示せ。(3) この制限は本節で提起された基本的な問題に対して満足のいく解を提供するだろうか。特に、数、レコード、バリアントなどの追加機能を伴う言語でも使いものになるだろうか。□

28.6　結びと交わり

16.3 節では、部分型付けの望ましい性質として任意の型の二つ組 S と T について**結び**が存在することを見た。つまり、S と T に共通するすべての上位型のうち最小の型 J である。この節では、カーネル $F_{<:}$ の部分型関係では任意の S と T について確かに結びが存在し、それだけでなく共通する部分型を少なくとも一つ持つような任意の S と T に関する交わり[†2]が存在することを、それらを計算するアルゴリズムを与えることにより示す。（一方、フル $F_{<:}$ に関してはこれらの性質が成り立たない。演習 28.6.3 を参照。）

$\Gamma \vdash S \vee T = J$ は「J は文脈 Γ において S と T の結びである」ことを、$\Gamma \vdash S \wedge T = M$ は「M は Γ において S と T の交わりである」ことを意味するものとする。これらの関係を計算するアルゴリズムは図 28-5 で同時に定義される。各定義において幾つかの場合分けが重複していることに注意されたい。この定義を決定的なアルゴリズムとして読むために、最初に適用できる節が常に選ばれるものとする。

∨ と ∧ が全域関数であることは容易に確認できる。∨ は必ず型を返し、∧ は型を返すか失敗する、という意味である。S と T の Γ に関する重みの合計（定義 28.3.4 を参照）が再帰呼び出しのたびに減少することを確認するだけである。

さて、これらの定義が実際に結びと交わりを計算することを検証しよう。議論は二つの部分に分かれる。命題 28.6.1 は、計算された結びは S と T の上界であることを、交わりは（存在すれば）下界であることを示している。そして命題 28.6.2 は、計算された結びは S と T に共通するすべての上界より小さいことを、交わりは共通するすべての下界よりも大きい（そして S と T の共通の下界があれば常に存在する）ことを示す。

[†2]（訳注）16.3 節の「有界な交わり」と同じ。

→ ∀ <: Top

$$
\Gamma \vdash S \vee T = \begin{cases}
T & \Gamma \vdash S <: T \text{ の場合} \\
S & \Gamma \vdash T <: S \text{ の場合} \\
J & S = X \\
 & X{<:}U \in \Gamma \\
 & \Gamma \vdash U \vee T = J \text{ の場合} \\
J & T = X \\
 & X{<:}U \in \Gamma \\
 & \Gamma \vdash S \vee U = J \text{ の場合} \\
M_1 \to J_2 & S = S_1 \to S_2 \\
 & T = T_1 \to T_2 \\
 & \Gamma \vdash S_1 \wedge T_1 = M_1 \\
 & \Gamma \vdash S_2 \vee T_2 = J_2 \text{ の場合} \\
\forall X{<:}U_1.J_2 & S = \forall X{<:}U_1.S_2 \\
 & T = \forall X{<:}U_1.T_2 \\
 & \Gamma, X{<:}U_1 \vdash S_2 \vee T_2 = J_2 \\
 & \quad \text{の場合} \\
Top & \text{それ以外の場合}
\end{cases}
$$

$$
\Gamma \vdash S \wedge T = \begin{cases}
S & \Gamma \vdash S <: T \text{ の場合} \\
T & \Gamma \vdash T <: S \text{ の場合} \\
J_1 \to M_2 & S = S_1 \to S_2 \\
 & T = T_1 \to T_2 \\
 & \Gamma \vdash S_1 \vee T_1 = J_1 \\
 & \Gamma \vdash S_2 \wedge T_2 = M_2 \text{ の場合} \\
\forall X{<:}U_1.M_2 & S = \forall X{<:}U_1.S_2 \\
 & T = \forall X{<:}U_1.T_2 \\
 & \Gamma, X{<:}U_1 \vdash S_2 \wedge T_2 = M_2 \\
 & \quad \text{の場合} \\
\mathit{fail} & \text{それ以外の場合}
\end{cases}
$$

▲ 図 28-5. カーネル $F_{<:}$ の結びと交わりを求めるアルゴリズム

命題 28.6.1.

(1) $\Gamma \vdash S \vee T = J$ ならば、$\Gamma \vdash S <: J$ かつ $\Gamma \vdash T <: J$

(2) $\Gamma \vdash S \wedge T = M$ ならば、$\Gamma \vdash M <: S$ かつ $\Gamma \vdash M <: T$ □

[証明] $\Gamma \vdash S \vee T = J$ または $\Gamma \vdash S \wedge T = M$ の導出のサイズ（つまり、J または M の計算に必要な再帰呼び出しの回数）に関する単純な帰納法による。 □

命題 28.6.2.

(1) $\Gamma \vdash S <: V$ かつ $\Gamma \vdash T <: V$ ならば、$\Gamma \vdash J <: V$ なる J が存在し $\Gamma \vdash S \vee T = J$

(2) $\Gamma \vdash L <: S$ かつ $\Gamma \vdash L <: T$ ならば、$\Gamma \vdash L <: M$ なる M が存在し $\Gamma \vdash S \wedge T = M$ □

[証明] (1) については $\Gamma \mapsto S <: V$ と $\Gamma \mapsto T <: V$ の、(2) については $\Gamma \mapsto L <: S$ と $\Gamma \mapsto L <: T$ の**アルゴリズム的**導出の合計サイズに関する帰納法により、二つを同時に証明するのが最も簡単である。(定理 28.3.3 により、ある導出について対応するアルゴリズム的導出が常に存在することが保証される。)

(1) もし二つの導出のどちらかが SA-Top のインスタンスならば、$V = Top$ であり、求めていた結果 $\Gamma \vdash J <: V$ が直ちに得られる。
もし $\Gamma \mapsto T <: V$ の導出が SA-Refl-TVar のインスタンスならば、$T = V$。しかし一つ目の導出によれば $\Gamma \vdash S <: V = T$ であるため、結びの定義における最初の節が適用でき、$\Gamma \vdash S \vee T = T$ が得られ要求が満たされる。同様に、$\Gamma \mapsto S <: V$ の導出が SA-Refl-TVar の

インスタンスならば、$\mathsf{S} = \mathsf{V}$。しかし二つ目の導出によれば $\Gamma \vdash \mathsf{T} <: \mathsf{V} = \mathsf{S}$ であるため、結びの定義における二つ目の節が適用でき、$\Gamma \vdash \mathsf{S} \vee \mathsf{T} = \mathsf{S}$ が得られやはり要求が満たされる。

$\Gamma \mapsto \mathsf{S} <: \mathsf{V}$ の導出が SA-TRANS-TVAR のインスタンスで終了するならば、$\mathsf{S} = \mathsf{X}$ かつ $\mathsf{X}<:\mathsf{U} \in \Gamma$ であり、$\Gamma \mapsto \mathsf{U} <: \mathsf{V}$ の部分導出がある。帰納法の仮定より、$\Gamma \vdash \mathsf{J} <: \mathsf{V}$ を満たすある J について、$\Gamma \vdash \mathsf{U} \vee \mathsf{T} = \mathsf{J}$ が成り立つ。結びの定義の三つ目の節より $\Gamma \vdash \mathsf{S} \vee \mathsf{T} = \mathsf{J}$ である。$\Gamma \mapsto \mathsf{T} <: \mathsf{V}$ の導出が SA-TRANS-TVAR で終了する場合も同様。

アルゴリズム的部分型付け規則の形から、残る可能性は、両方の導出が SA-ARROW で終了するか、SA-ALL で終了するかのどちらかであることは容易に確認できる。

もし両方が SA-ARROW で終了しているならば、$\mathsf{S} = \mathsf{S}_1 \to \mathsf{S}_2$、$\mathsf{T} = \mathsf{T}_1 \to \mathsf{T}_2$、かつ $\mathsf{V} = \mathsf{V}_1 \to \mathsf{V}_2$ であり、ここで $\Gamma \mapsto \mathsf{V}_1 <: \mathsf{S}_1$、$\Gamma \mapsto \mathsf{S}_2 <: \mathsf{V}_2$、$\Gamma \mapsto \mathsf{V}_1 <: \mathsf{T}_1$、かつ $\Gamma \mapsto \mathsf{T}_2 <: \mathsf{V}_2$ である。帰納法の仮定の (2) より、$\Gamma \vdash \mathsf{V}_1 <: \mathsf{M}_1$ なる M_1 について $\Gamma \vdash \mathsf{S}_1 \wedge \mathsf{T}_1 = \mathsf{M}_1$ であり、(1) より、$\Gamma \vdash \mathsf{J}_2 <: \mathsf{V}_2$ なる J_2 について $\Gamma \vdash \mathsf{S}_2 \vee \mathsf{T}_2 = \mathsf{J}_2$ である。結びの定義の五番目の節より $\Gamma \vdash \mathsf{S}_1 \to \mathsf{S}_2 \vee \mathsf{T}_1 \to \mathsf{T}_2 = \mathsf{M}_1 \to \mathsf{J}_2$ であり、S-ARROW より $\Gamma \vdash \mathsf{M}_1 \to \mathsf{J}_2 <: \mathsf{V}_1 \to \mathsf{V}_2$ が得られる。

最後に、両方の導出が SA-ALL ならば、$\mathsf{S} = \forall \mathsf{X}<:\mathsf{U}_1.\mathsf{S}_2$、$\mathsf{T} = \forall \mathsf{X}<:\mathsf{U}_1 \to \mathsf{T}_2$ かつ $\mathsf{V} = \forall \mathsf{X}<:\mathsf{U}_1.\mathsf{V}_2$ であり、ここで $\Gamma, \mathsf{X}<:\mathsf{U}_1 \mapsto \mathsf{S}_2 <: \mathsf{V}_2$ かつ $\Gamma, \mathsf{X}<:\mathsf{U}_1 \mapsto \mathsf{T}_2 <: \mathsf{V}_2$ である。帰納法の仮定の (1) より、$\Gamma, \mathsf{X}<:\mathsf{U}_1 \vdash \mathsf{S}_2 \vee \mathsf{T}_2 = \mathsf{J}_2$ であり、ここで $\Gamma, \mathsf{X}<:\mathsf{U}_1 \mapsto \mathsf{J}_2 <: \mathsf{V}_2$。結びの定義の六番目の節より $\mathsf{J} = \forall \mathsf{X}<:\mathsf{U}_1.\mathsf{J}_2$ であり、S-ALL より $\Gamma \vdash \forall \mathsf{X}<:\mathsf{U}_1.\mathsf{J}_2 <: \forall \mathsf{X}<:\mathsf{U}_1.\mathsf{V}_2$ が得られる。

(2) $\Gamma \mapsto \mathsf{L} <: \mathsf{T}$ の導出が SA-TOP で終わるならば T は Top、このため $\Gamma \vdash \mathsf{S} <: \mathsf{T}$ であり、交わりの定義の一つ目の節より、$\Gamma \vdash \mathsf{S} \wedge \mathsf{T} = \mathsf{S}$ である。しかし、もう一方の導出によれば、$\Gamma \vdash \mathsf{L} <: \mathsf{S}$ であるため、これで完了である。$\Gamma \mapsto \mathsf{L} <: \mathsf{S}$ が SA-TOP で終わる場合も同様である。

$\Gamma \mapsto \mathsf{L} <: \mathsf{S}$ が SA-REFL-TVAR で終わるならば、$\mathsf{L} = \mathsf{S}$ かつもう一方の導出から $\Gamma \vdash \mathsf{L} = \mathsf{S} <: \mathsf{T}$ がわかり、これより交わりの定義から $\Gamma \vdash \mathsf{S} \wedge \mathsf{T} = \mathsf{S}$ であり、これで完了である。$\Gamma \mapsto \mathsf{L} <: \mathsf{T}$ の導出が SA-REFL-TVAR で終わる場合も同様である。

残った可能性は両方の導出が SA-TRANS-TVAR、SA-ARROW か SA-ALL で終了する場合である。

もし両方の導出が SA-TRANS-TVAR で終わるならば、$\mathsf{L} = \mathsf{X}$ であり、ここで $\mathsf{X}<:\mathsf{U} \in \Gamma$ かつ二つの部分導出 $\Gamma \mapsto \mathsf{U} <: \mathsf{S}$ と $\Gamma \mapsto \mathsf{U} <: \mathsf{T}$ が得られる。帰納法の仮定の (2) より $\Gamma \vdash \mathsf{U} <: \mathsf{M}$ であり、これより S-TVAR と推移律から $\Gamma \vdash \mathsf{L} <: \mathsf{M}$ が得られる。

もし両方の導出が SA-ARROW で終了するならば、$\mathsf{S} = \mathsf{S}_1 \to \mathsf{S}_2$、$\mathsf{T} = \mathsf{T}_1 \to \mathsf{T}_2$ かつ $\mathsf{L} = \mathsf{L}_1 \to \mathsf{L}_2$ であり、ここで $\Gamma \mapsto \mathsf{S}_1 <: \mathsf{L}_1$、$\Gamma \mapsto \mathsf{L}_2 <: \mathsf{S}_2$、$\Gamma \mapsto \mathsf{T}_1 <: \mathsf{L}_1$ かつ $\Gamma \mapsto \mathsf{L}_2 <: \mathsf{T}_2$。帰納法の仮定の (1) より、$\Gamma \vdash \mathsf{J}_1 <: \mathsf{L}_1$ なる J_1 について $\Gamma \vdash \mathsf{S}_1 \vee \mathsf{T}_1 = \mathsf{J}_1$ であり、さらに (2) より、$\Gamma \vdash \mathsf{L}_2 <: \mathsf{M}_2$ なる M_2 について $\Gamma \vdash \mathsf{S}_2 \wedge \mathsf{T}_2 = \mathsf{M}_2$ である。交わりの定義より $\Gamma \vdash \mathsf{S}_1 \to \mathsf{S}_2 \wedge \mathsf{T}_1 \to \mathsf{T}_2 = \mathsf{J}_1 \to \mathsf{M}_2$ であり、S-ARROW より $\Gamma \vdash \mathsf{L}_1 \to \mathsf{L}_2 <: \mathsf{J}_1 \to \mathsf{M}_2$ が得られる。

両方の導出が SA-ALL である場合も同様である。□

演習 28.6.3.［推奨, ★★★］　型の二つ組 $\mathsf{S} = \forall \mathsf{X}<:\mathsf{Y} \to \mathsf{Z}.\mathsf{Y} \to \mathsf{Z}$ と $\mathsf{T} = \forall \mathsf{X}<:\mathsf{Y}' \to \mathsf{Z}'.\mathsf{Y}' \to \mathsf{Z}'$ （Ghelli [171] による）と、文脈 $\Gamma = \mathsf{Y}<:\mathsf{Top}, \mathsf{Z}<:\mathsf{Top}, \mathsf{Y}'<:\mathsf{Y}, \mathsf{Z}'<:\mathsf{Z}$ を考える。(1) フル $F_{<:}$ では、Γ の下で S と T の両方の部分型である型は幾つあるだろうか。(2) フル $F_{<:}$ では、型 S と T は Γ の下で交わりを持たないことを示せ。(3) フル $F_{<:}$ において、Γ の下で結びを持たない型の二つ組を探せ。□

28.7 有界存在量化

カーネル $F_{<:}$ の型検査アルゴリズムを、存在型を持つ言語に拡張するには、さらに繊細に扱わねばならないことがある。存在型のための宣言的な除去規則を思い出そう。

$$\frac{\Gamma \vdash \mathtt{t}_1 : \{\exists \mathtt{X}\text{<:}\mathtt{T}_{11},\mathtt{T}_{12}\} \qquad \Gamma, \mathtt{X}\text{<:}\mathtt{T}_{11}, \mathtt{x}:\mathtt{T}_{12} \vdash \mathtt{t}_2 : \mathtt{T}_2}{\Gamma \vdash \mathtt{let}\ \{\mathtt{X},\mathtt{x}\}\text{=}\mathtt{t}_1\ \mathtt{in}\ \mathtt{t}_2 : \mathtt{T}_2} \quad \text{(T-UNPACK)}$$

288 ページで、型変数 X は二番目の前提で $\mathtt{t}_2$ の型が計算される文脈において現れるのであり、規則の結論の文脈には**現れない**ことを述べた。これは、どの自由な X の出現も結論においてはスコープ外であるため、型 $\mathtt{T}_2$ は自由な X を含んではならないことを意味する。この点については 25.5 節でより詳細に議論したが、そこでは、前提から結論への文脈の変化が、型を名無しの de Bruijn 形式で表現した場合の、$\mathtt{T}_2$ 中の変数のインデックスを負の方向へシフトすることに対応することを述べた。このシフトは $\mathtt{T}_2$ が自由な X を含んでいるとき失敗する。

存在型を持つ言語のための最小型付けアルゴリズムに関するこの観察から、何がわかるだろうか？ 特に、t = let {X,x} = p in x のような式において、p が型 {∃X,Nat→X} を持つような場合にはどうするべきだろうか？ 本体の x の最も自然な型は Nat→X である。これは束縛された型変数 X に言及している。しかしながら、(包摂規則付きの) 宣言的な型付け関係によれば、x は型 Nat→Top や型 Top も持つ。どちらも X に言及していないので、宣言的な型システムでは項 t の全体に型 Nat→Top や型 Top を合法的に割り当てられる。より一般的には、アンパッケージ化の式の本体をいつでも、束縛された型変数 X を伴わない型に昇格し、それから T-UNPACK を適用してよい。このため、最小型付けのアルゴリズムを完全にしたいならば、本体の最小型 $\mathtt{T}_2$ が束縛変数 X の自由な出現を含むようなアンパッケージ化式を見つけても、単に失敗するべきではない。その代わり、X に言及しない何らかの上位型へ $\mathtt{T}_2$ を昇格することを試みるべきである。これを動作させるために鍵となるのは、ある型について X を含まない上位型の集合が常に最小の要素を持つという観察であり、次の演習（解は Ghelli and Pierce [175] による）で示される。

演習 28.7.1. [★★★] 有界存在型を持つカーネル $F_{<:}$ において、ある文脈 Γ における型 T について X が出現しない最小の上位型を計算するアルゴリズム $R_{\mathtt{X},\Gamma}(\mathtt{T})$ を与えよ。 □

存在型の除去のためのアルゴリズム的型付け規則はこれを用いて次のように書ける。

$$\frac{\begin{array}{c}\Gamma \mapsto \mathtt{t}_1 : \mathtt{T}_1 \qquad \Gamma \mapsto \mathtt{T}_1 \Uparrow \{\exists \mathtt{X}\text{<:}\mathtt{T}_{11},\mathtt{T}_{12}\} \\ \Gamma, \mathtt{X}\text{<:}\mathtt{T}_{11}, \mathtt{x}:\mathtt{T}_{12} \mapsto \mathtt{t}_2 : \mathtt{T}_2 \qquad R_{\mathtt{X},(\Gamma,\mathtt{X}<:\mathtt{T}_{11},\mathtt{x}:\mathtt{T}_{12})}(\mathtt{T}_2) = \mathtt{T}_2'\end{array}}{\Gamma \mapsto \mathtt{let}\ \{\mathtt{X},\mathtt{x}\}\text{=}\mathtt{t}_1\ \mathtt{in}\ \mathtt{t}_2 : \mathtt{T}_2'} \quad \text{(TA-UNPACK)}$$

有界存在型を持つ**フル** $F_{<:}$ については、予想通り状況はより厄介である。Ghelli and Pierce [175] は、Γ の元での、T の X を含まない上位型の集合が、最小の要素を持たないような型 T、文脈 Γ、変数 X の例を与えている。直ちに、この型システムの型付け関係は最小型を持たないことがわかる。

演習 28.7.2. [★★★] フル $F_{<:}$ の変種で有界存在型のみを持つもの（全称型がない）の部分型関係もまた決定不能であることを示せ。 □

28.8 有界量化と Bottom 型

最小の Bot 型（15.4 節）を追加すると、$F_{<:}$ のメタ理論的性質はいくぶん複雑になる。なぜなら、∀X<:Bot.T の形を持つ型において、変数 X は仮定より Bot の部分型であり、規則 S-Bot より Bot は X の部分型であるため、X が T の中で実質的に Bot の**別称**になっているからである。このことから、∀X<:Bot.X→X と ∀X<:Bot.Bot→Bot のような型の二つ組が部分型関係において同値であることになるが、これらは構文的に同一でない。さらに、もし周囲の文脈が仮定 X<:Bot と Y<:Bot を含むならば、型 X→Y と Y→X はどちらも Bot に直接言及していないにもかかわらず同値になる。このような困難にもかかわらず、カーネル $F_{<:}$ に重要な性質は Bot が存在しても成り立つ。詳細は Pierce [368] にある。

第VI部

高階の型システム

型演算子とカインド

本章で導入する体系は、型演算子を持つ純粋な単純型付きラムダ計算 λ_ω（図 29-1）である。例では数やブール値（図 8-2）、全称型（図 23-1）も用いる。対応する OCaml 実装は fullomega である。

ここまでの章ではしばしば、例を読みやすくするために、

```
CBool = ∀X. X → X → X;
```

や

```
Pair Y Z = ∀X. (Y→Z→X) → X;
```

のような省略を用いてきた。これにより例えば、λx:∀X.(Nat→Bool→X)→X. x の代わりに、より簡潔に λx:Pair Nat Bool. x と書ける。

CBool は単なる省略である。例の中に CBool があったら、単純にその定義の右辺で置き換えるべきである。一方 Pair は、**パラメータ付き省略**である。Pair S T があれば、その定義の中の仮引数 Y と Z に対し、実際の型 S と T をそれぞれ代入しなければならない。つまり Pair のような省略は、型式レベルの関数を定義するための非形式的な記法を与えている。

型構築子 Array や Ref を含む型レベルの式（Array T や Ref T など）も用いてきた。これらの型構築子は、プログラマが定義するのではなく言語組み込みのものであるが、これらもまた型レベルの関数の一種である。例えば Ref は、各型 T に対して、T の要素を持つ参照セルの型を返す関数とみなせる。

本章および第 30 章と第 31 章では、これら型レベルの関数（まとめて**型演算子**と呼ぶ）をより形式的に扱う。本章では、型レベルの抽象と適用の基本的な仕組みを導入する。併せて、二つの型式を等価とみなすべき条件を正確に定義する。さらに、型の形式の正しさを表す関係、すなわち**カインド付け**関係を導入し、無意味な型式を書いてしまうことを防ぐ。第 30 章ではさらに踏み込んで、型演算子を**一級市民**（すなわち、引数として関数に渡せるもの）として扱い、System F_ω というよく知られた体系を導入する。これは System F における型に関する量化（第 23 章）を一般化して、型演算子に関する**高階量化**にしたものである。第 31 章では、型演算子と高階量化と部分型の組み合わせを考える。

29.1 直観

型レベルの関数を考える上でまず必要になるのは、抽象と適用の表記法である。項レベルの抽象や適用と**同じ**記法を転用するのが慣例となっている。すなわち、抽象は λ で、適用は並置で示す[†1]。例えば、型 T を受け取り、レコード型 {a:T,b:T} を返す関数を λX.{a:X,b:X} と書く。この関数を引数 Bool に適用することは (λX.{a:X,b:X}) Bool と書く。

通常の関数と同じように、複数の引数を取る型関数は、**カリー化**によって 1 引数関数から組み上げることができる。例えば、型関数 λY. λZ. ∀X. (Y→Z→X) → X は 2 引数関数を表現している。

[†1] このように記法を節約することの欠点は、異なる種類の式を表す用語が少し入り乱れてしまうことである。特に「型抽象」というフレーズは、引数として型を期待するラムダ抽象（すなわち、λX.t のような項）を指すかもしれないし、型レベルの抽象（すなわち、λX.{a:X} のような型式）を指すかもしれない。どちらもありうる文脈では、前者の意味に対しては「多相関数」と呼び、後者の意味に対しては「型レベル抽象」や「演算子抽象」と呼ぶ傾向がある。

これは厳密にいえば 1 引数関数だが、型 S に適用すると別の 1 引数関数を返し、それを型 T に適用すると型 ∀X.(S→T→X)→X を返す。

型演算子を含め、長い型式に対し、今まで通り非形式的省略を使用する。例えば、以後本章では、次の省略を使うこととする。

```
Pair = λY. λZ. ∀X. (Y→Z→X) → X;
```

例において Pair S T と書かれていたら、実際には以下の意味である。

```
(λY. λZ. ∀X. (Y→Z→X) → X) S T
```

言い換えると、これまで用いてきたパラメータ付き省略の非形式的慣習をやめ、より簡単な非形式的慣習として、単純な省略が出てくるごとに定義の右辺で展開することとする。また、形式的な仕組みにより、さらに型演算子を定義したり具体化したりする。省略の定義や展開の演算も形式的に扱うことはできる（すなわち、メタ言語における慣習ではなく、対象言語の演算にできる）が、ここではそのようにしない。興味のある読者は、**定義**や**一元カインド**のある型システムに関する文献を参照するとよい。Severi and Poll [431]、Stone and Harper [445]、Crary [121] と、そこで参照されている研究を見よ。

型レベルの抽象と適用を導入すると、同じ型をさまざまな方法で書ける可能性が生まれる。例えば、Id が型演算子 λX.X の省略だとしたら、以下の式はすべて同じ矢印型を表す名前である。

Nat → Bool	Nat → Id Bool	Id Nat → Id Bool
Id Nat → Bool	Id (Nat → Bool)	Id (Id (Id Nat → Bool))

この直観を正確にするために、型の定義による等価関係を導入する。これを S ≡ T と書く。この関係の定義において最も重要な規則は以下である。

$$(\lambda X{::}K_{11}.T_{12})\ T_2 \equiv [X \mapsto T_2]T_{12} \qquad \text{(Q-AppAbs)}$$

この規則は、型レベルの抽象を実引数に適用したものは、抽象の本体の仮引数に実引数を代入したものと等価であるということを述べている。型検査で定義による等価性を利用するには、以下の新しい規則を用いる。

$$\frac{\Gamma \vdash t : S \qquad S \equiv T}{\Gamma \vdash t : T} \qquad \text{(T-Eq)}$$

この規則は、二つの型が等価であれば、一方の要素はすべてもう一方の要素であるという直観を正確に捉えている。

抽象と適用の仕組みから生まれるもう一つの新しい可能性は、無意味な型式を書けてしまう能力である。例えば、真の型を真の型に適用すること（つまり (Bool Nat) という型式など）は、項レベルで true を 6 に適用するのと同じく、意味をなさない。このような無意味な式を防ぐべく、**カインド**のシステムを導入する。カインドは、矢印型から項の引数の数がわかるのと同様に、型式を引数の数によって分類する。

カインドは、一つの原子カインド（* と書き、「型」と読む）と、一つの構築子 ⇒ から構築される。例えば以下がカインドである。

*	真の型のカインド（Bool や Bool→Bool など）
⇒	型演算子のカインド（つまり真の型から真の型への関数）
⇒⇒*	真の型から型演算子への関数のカインド（つまり 2 引数の型演算子）
(*⇒*)⇒*	型演算子から真の型への関数のカインド

つまりカインドは「型の型」である。カインドのシステムは本質的に、単純型付きラムダ計算を「1レベル上」に複製したものである。

以下では**型**という単語を、型レベルの任意の式に対して用いる。つまり Nat→Nat や ∀X.X→X のような通常の型と、λX.X のような型演算子を、どちらも型と呼ぶ。通常の型（つまり実際に項を分類するための型式）に限定したいときは、**真の型**という。

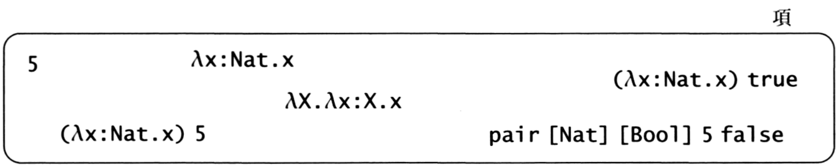
(*⇒*)⇒* のようなカインドを持つ型式を、**高階型演算子**と呼ぶ。項レベルの高階関数はしばしばとても有用であるが、これに対し高階型演算子はやや難解である。第 32 章で高階型演算子を用いるある種の用例を取り上げる。

型式が正しくカインド付けされているか検査する問題を簡単にするため、型レベルの抽象の束縛変数すべてに対し、カインドを注釈することにする。例えば Pair 演算子の正式な形は以下のようになる。

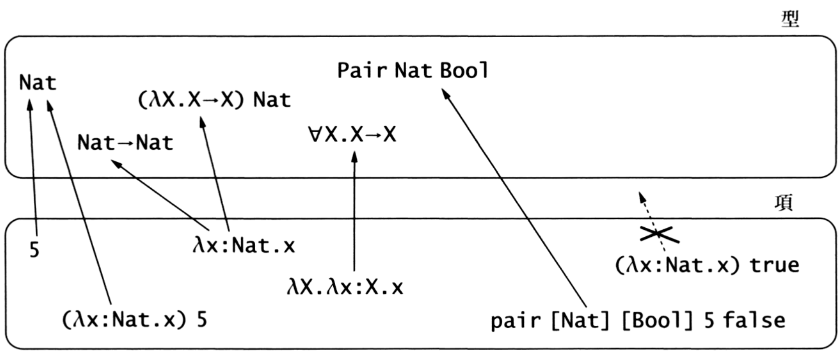
```
Pair = λA::*. λB::*. ∀X.  (A→B→X) → X;
```

（二重コロンであることに注意されたい。）しかし、このような型注釈はほとんどすべて * なので、今まで通り λX.T と書いたら、λX::*.T の省略であることにする。

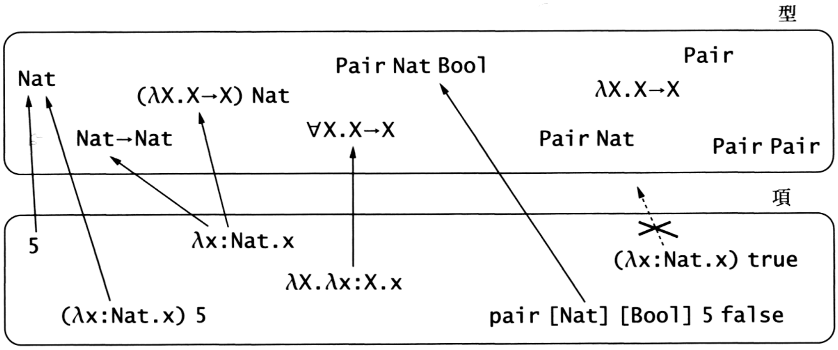
図を幾つか見れば、理解が明確になるだろう。ここで我々の言語の式を、項、型、カインドの三つのクラスに分ける。項レベルは、基本データ型（整数や浮動小数）、複合的なデータ型（レコードなど）、値レベルの抽象と適用、型抽象と型適用からなる。



型レベルは二種類の式からなる。一つは Nat、Nat→Nat、Pair Nat Bool、∀X.X→X のような真の型である。その要素は項である。（もちろん、すべての項が型を持つわけではない。例えば (λx:Nat.x) true は型を持たない。）

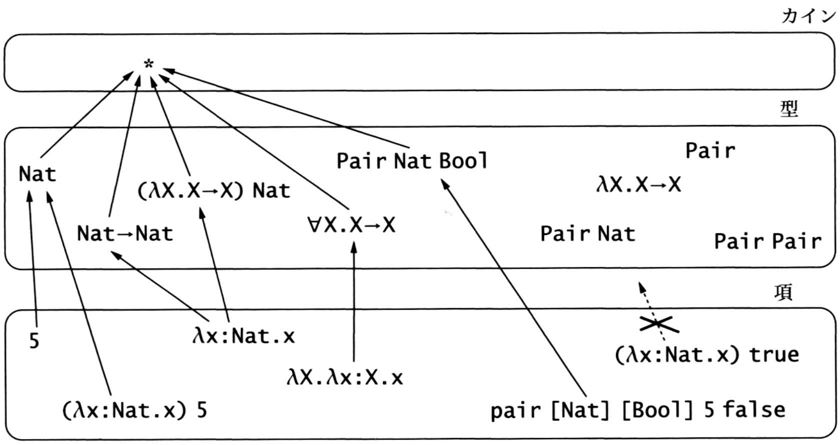


もう一つは、Pair や λX.X→X のような型演算子である。型演算子自体は項を分類**しない**（「型 λX.X→X を持つ項とはどんなものか？」と尋ねるのは意味をなさない）。しかし型演算子は型引数に適用することで、(λX.X→X)Nat のような、実際に項を分類する真の型を作れる。

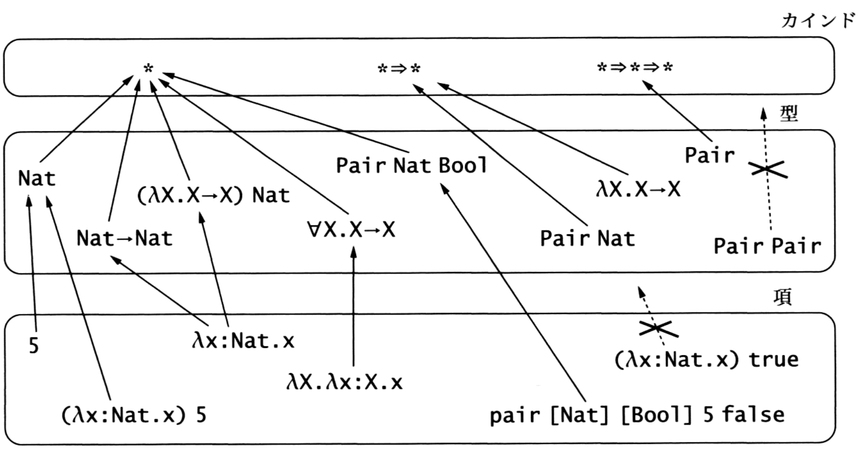


真の型（つまりカインド * の型式）は、(λX.X→X) Nat や Pair Nat Bool などのように、高階カインドの型演算子を一部に含むかもしれない。これは、(λx:Nat.x) 5 のように、Nat などの基本型に属す項の式が、部分式としてラムダ抽象を含むのと同じである。

最後に、カインドレベルがある。最も単純なカインドは * であり、これは真の型すべてを要素とする。



λX.X→X と Pair などの型演算子は *⇒* や *⇒*⇒* といった矢印カインドに所属する。正しい形式を持たない型レベルの式（例えば Pair Pair）は、どのようなカインドにも属さない。



演習 29.1.1. [★] 型レベルの式 ∀X.X→X と λX.X→X の意味は何が異なるか。 □

演習 29.1.2. [★] Nat→Nat のような矢印型が *⇒* といった矢印カインドを持たないのはなぜだろうか。 □

ここで「式は 3 レベルで終わりなのか？」と考えるのは自然である。カインドからカインドへの関数やカインドレベルでの適用などを導入し、カインドの式をその機能に従って分類する第四のレベルを追加する、ということを**無限に**続けることは無理なのだろうか？ そのようなシステムは、**純粋型システム**のコミュニティ（Terlouw [458]; Berardi [43]; Barendregt [39; 40]; Jutting, McKinna, and Pollack [245]; McKinna and Pollack [305]; Pollack [390]）で研究されている。しかしプログラミング言語においては、3 レベルで十分だとわかっている。

実際、型演算子は基本的にすべての静的型付けのプログラミング言語において何らかの形で存在するものであるが、本章のような 3 レベルの体系の全能力すら、言語設計者がプログラマに提供することは比較的まれである。一部の言語（例えば Java）は、Array など組み込みの型演算子を幾つか提供しているだけで、新しい型演算子を定義する機能は提供していない[†2]。他の言語では、型演算子を他の機能とまとめて一つの機能としているものもある。例えば ML では、型演算子は datatype の仕組みの一部として提供されている。これにより、以下のような**パラメータ付きデータ型**を定義できる[†3]。

```
type 'a Tyop = tyoptag of ('a → 'a)
```

我々はこれを以下のように書いている。

```
Tyop = λX. <tyoptag:X→X>;
```

つまり ML では、任意のパラメータ型ではなく、パラメータ付きのバリアント型だけを定義できる。このように制限することの利点は、型レベルで型演算子 Tyop がプログラムに現れるときには常に、対応するタグ tyoptag が項レベルで現れることである。つまり、型検査器が定義による等価性を用いて、Tyop Nat のような型を Nat→Nat といった簡約済みの形に置き換える必要がある箇所はすべて、プログラム中に tyoptag が現れることで明示される。これにより型検査アルゴリズムは相当簡単になる[†4]。

カインドの ⇒ 構築子は、ここで紙面を割いて議論する唯一の構築子であるが、他にも非常に多くの構築子が研究されている。実際、型式のさまざまな性質を検査・追跡するためのカインドのシステムにはさまざまなものがあり、項の性質を解析する型システムの多様さに匹敵する。**レコードのカインド**（要素が型のレコード。レコードの型と混同しないこと。相互再帰する型の体系を定義する自然な方法を提供する）、**列カインド**（「フィールドの列」を表す。これを用いると、列変数多相を持つ体系においてレコード型を組み立てることができる。265 ページを見よ）、**冪カインド**または**冪型**（部分型の別の形式化を与える。Cardelli [76] を見よ）、**一元カインド**（348 ページで説明した**定義**に関連する。また、367 ページで説明する**シェアリング**を持つモジュールシステムにも関連する）、**依存カインド**（30.5 節で議論する依存型の「1 レベル上」の類似物）、他多数の研究が存在する。

[†2] （訳注）Java 5.0 で総称型が導入された。

[†3] 例をわかりやすくするため、ここでは識別子における大文字・小文字の使い方に関する ML の慣習を無視している。OCaml でこの定義は、本当は以下のように書く。

```
type 'a tyop = Tyoptag of ('a → 'a)
```

[†4] この制限は、217 ページで議論した ML における再帰型の扱いに似ている。再帰型を datatype 定義と組み合わせると、そのバリアント型に関連付けられたタグ付けや case 分岐の演算に fold/unfold の注釈を隠すことで、プログラマは同値再帰型の利便性を得られるし、型検査器は同型再帰型の簡単さを得られる。

29.2　定義

図 29-1 は型演算子を持つ核となるラムダ計算の完全な定義を与える。項レベルでは、この計算は単純型付きラムダ計算における変数、抽象、適用だけを持つ（このためこれは、**型演算子を持つ単純型付きラムダ計算**と呼ばれる）。型レベルには、通常の矢印型や型変数に加え、演算子抽象と演算子適用がある。∀X.T のような量化型はこの体系では省略する。量化型については第 30 章で改めて詳しく考える。

この体系の形式化は、単純型付きラムダ計算の仕組みに三つの拡張を行っている。一つ目は、**カインド付け**規則の集まりを加えることである。これは、新しい型式を作るためにどのように型式を組み合わせられるかを規定する。$\Gamma \vdash$ T :: K と書くと、「文脈 Γ において型 T はカインド K を持つ」ということを表す。これらのカインド付け規則と、元の単純型付きラムダ計算の型付け規則（図 9-1）が似ていることに注意されたい。

二つ目は、（λx:T.t のように）型 T が項に現れるとき、T が正しい形式を持つことを検査しなければならないことである。このため、以前からある T-Abs 規則に $\Gamma \vdash$ T :: * を検査する前提を新たに追加している。T がちょうどカインド * を持つ（つまり T は真の型でなければならない）ことに注意されたい。なぜなら T は、項変数 x がどういう値の上を動くかを記述するために用いられているからである。型付け規則は次の不変条件を保つ。すなわち、型付け判断式 $\Gamma \vdash$ t : T を導出できるときはいつでも、カインド付け判断式 $\Gamma \vdash$ T :: * も導出できる（ただし文脈に現れる型がすべて正しくカインド付けされている限り）。このことは 363 ページでもっと詳しく議論する。

三つ目は、型の**定義による等価**関係を表す一連の規則を加えることである。S ≡ T と書くと、「型 S と T は定義により等価である」ということを表す。この関係は項レベルの簡約関係によく似ている。定義による等価性が型付けに関して与える影響は、T-Eq という新たな規則が捉えている。カインド付けの前提（前節でこの規則を議論したときには省略していた）は、上で触れた不変条件、つまり「型付け可能な項は常にカインド付け可能な型を持つ」ということを維持している。この規則と、部分型付けのある型システムの包摂規則（T-Sub）が似ていることに注意されたい。

この体系の基本的なメタ理論的性質を説明するには、少し手間がかかる。型等価関係が、項に割り当てられる型の「外見」を非常に柔軟なものにしてしまうためである。この理論の説明は第 30 章に持ち越す。

→ ⇒ λ→（図 9-1）の拡張

構文

t ::=		項：
	x	変数
	$\lambda x{:}T.t$	ラムダ抽象
	t t	関数適用
v ::=		値：
	$\lambda x{:}T.t$	ラムダ抽象値
T ::=		型：
	X	型変数
	$\lambda X{::}K.T$	演算子抽象
	T T	演算子適用
	$T\to T$	関数の型
Γ ::=		文脈：
	∅	空の文脈
	Γ, x:T	項変数の束縛
	Γ, X::K	型変数の束縛
K ::=		カインド：
	*	真の型のカインド
	K⇒K	演算子のカインド

評価 $t \longrightarrow t'$

$$\frac{t_1 \longrightarrow t_1'}{t_1\ t_2 \longrightarrow t_1'\ t_2} \quad \text{(E-APP1)}$$

$$\frac{t_2 \longrightarrow t_2'}{v_1\ t_2 \longrightarrow v_1\ t_2'} \quad \text{(E-APP2)}$$

$$(\lambda x{:}T_{11}.t_{12})\ v_2 \longrightarrow [x \mapsto v_2]t_{12} \quad \text{(E-APPABS)}$$

カインド付け $\Gamma \vdash T :: K$

$$\frac{X{::}K \in \Gamma}{\Gamma \vdash X :: K} \quad \text{(K-TVAR)}$$

$$\frac{\Gamma, X{::}K_1 \vdash T_2 :: K_2}{\Gamma \vdash \lambda X{::}K_1.T_2 :: K_1 \Rightarrow K_2} \quad \text{(K-ABS)}$$

$$\frac{\Gamma \vdash T_1 :: K_{11} \Rightarrow K_{12} \qquad \Gamma \vdash T_2 :: K_{11}}{\Gamma \vdash T_1\ T_2 :: K_{12}} \quad \text{(K-APP)}$$

$$\frac{\Gamma \vdash T_1 :: * \qquad \Gamma \vdash T_2 :: *}{\Gamma \vdash T_1 \to T_2 :: *} \quad \text{(K-ARROW)}$$

型等価関係 $S \equiv T$

$$T \equiv T \quad \text{(Q-REFL)}$$

$$\frac{T \equiv S}{S \equiv T} \quad \text{(Q-SYMM)}$$

$$\frac{S \equiv U \qquad U \equiv T}{S \equiv T} \quad \text{(Q-TRANS)}$$

$$\frac{S_1 \equiv T_1 \qquad S_2 \equiv T_2}{S_1 \to S_2 \equiv T_1 \to T_2} \quad \text{(Q-ARROW)}$$

$$\frac{S_2 \equiv T_2}{\lambda X{::}K_1.S_2 \equiv \lambda X{::}K_1.T_2} \quad \text{(Q-ABS)}$$

$$\frac{S_1 \equiv T_1 \qquad S_2 \equiv T_2}{S_1\ S_2 \equiv T_1\ T_2} \quad \text{(Q-APP)}$$

$$(\lambda X{::}K_{11}.T_{12})\ T_2 \equiv [X \mapsto T_2]T_{12} \quad \text{(Q-APPABS)}$$

型付け $\Gamma \vdash t : T$

$$\frac{x{:}T \in \Gamma}{\Gamma \vdash x : T} \quad \text{(T-VAR)}$$

$$\frac{\Gamma \vdash T_1 :: * \qquad \Gamma, x{:}T_1 \vdash t_2 : T_2}{\Gamma \vdash \lambda x{:}T_1.t_2 : T_1 \to T_2} \quad \text{(T-ABS)}$$

$$\frac{\Gamma \vdash t_1 : T_{11} \to T_{12} \qquad \Gamma \vdash t_2 : T_{11}}{\Gamma \vdash t_1\ t_2 : T_{12}} \quad \text{(T-APP)}$$

$$\frac{\Gamma \vdash t : S \qquad S \equiv T \qquad \Gamma \vdash T :: *}{\Gamma \vdash t : T} \quad \text{(T-EQ)}$$

▲図 29-1. 型演算子とカインド付け（λ_ω）

第 30 章

高階多相

本章の例は、レコード、ブール値、存在型（図 30-2）を持つ F_ω（図 30-1）の項である。対応する OCaml 実装は `fullomega` である。30.5 節で触れる依存型の実装は提供していない。

第 29 章では $\lambda_{\rightarrow}$ に型演算子を加える方法を見てきた。ごく自然な次のステップは、本書を通して学んできたさまざまな型付け機能を型演算子と組み合わせることである。本章では、型演算子と System F の多相性とを組み合わせ、F_ω（Girard [177]）と呼ばれるよく知られた体系を構成する。第 31 章では、この体系を部分型付けで拡張して System $F_{<:}^{\omega}$ を構成する。System $F_{<:}^{\omega}$ は、第 32 章において純粋関数的オブジェクトに関する最後の事例で用いる体系である。

F_ω の定義は、λ_ω の機能と System F の機能を単純に組み合わせたものである。しかし、この体系の基本的な性質（特に保存と進行）の証明は、これまで見てきたほとんどの体系より多少大変である。これは、型検査器が型レベルの評価を必要とすることに対処しなければならないためである。この証明が本章の主題となる。

30.1 定義

System F_ω は第 23 章の System F と第 29 章の λ_ω を組み合わせて構成される。型変数が束縛される箇所（つまり型抽象と型量化子）に、カインド付け注釈（X::K）が追加される。全称量化子だけを含む（存在量化子は持たない）体系の形式的定義を図 30-1 に示す。30.3 節の証明で参照しやすいよう、以前の体系との差分は小さいが省略せず規則を列挙している。

→ ∀ ⇒　　　　λ_ω（図 29-1）と System F（図 23-1）の拡張

構文

t ::=		項：
	x	変数
	λx:T.t	ラムダ抽象
	t t	関数適用
	λX::K.t	型抽象
	t [T]	型適用
v ::=		値：
	λx:T.t	ラムダ抽象値
	λX::K.t	型抽象値
T ::=		型：
	X	型変数
	T→T	関数の型
	∀X::K.T	全称型
	λX::K.T	演算子抽象
	T T	演算子適用
Γ ::=		文脈：
	∅	空の文脈
	Γ, x:T	項変数の束縛
	Γ, X::K	型変数の束縛
K ::=		カインド：
	*	真の型のカインド
	K⇒K	演算子のカインド

続く . . .

▲ 図 30-1.　高階多相ラムダ計算（F_ω）

評価 $\boxed{t \longrightarrow t'}$

$$\frac{t_1 \longrightarrow t_1'}{t_1\ t_2 \longrightarrow t_1'\ t_2} \quad \text{(E-App1)}$$

$$\frac{t_2 \longrightarrow t_2'}{v_1\ t_2 \longrightarrow v_1\ t_2'} \quad \text{(E-App2)}$$

$$(\lambda x{:}T_{11}.t_{12})\ v_2 \longrightarrow [x \mapsto v_2]t_{12} \quad \text{(E-AppAbs)}$$

$$\frac{t_1 \longrightarrow t_1'}{t_1\ [T_2] \longrightarrow t_1'\ [T_2]} \quad \text{(E-TApp)}$$

$$(\lambda X{::}K_{11}.t_{12})\ [T_2] \longrightarrow [X \mapsto T_2]t_{12} \quad \text{(E-TappTabs)}$$

カインド付け $\boxed{\Gamma \vdash T :: K}$

$$\frac{X{::}K \in \Gamma}{\Gamma \vdash X :: K} \quad \text{(K-TVar)}$$

$$\frac{\Gamma, X{::}K_1 \vdash T_2 :: K_2}{\Gamma \vdash \lambda X{::}K_1.T_2 :: K_1 \Rightarrow K_2} \quad \text{(K-Abs)}$$

$$\frac{\Gamma \vdash T_1 :: K_{11} \Rightarrow K_{12} \qquad \Gamma \vdash T_2 :: K_{11}}{\Gamma \vdash T_1\ T_2 :: K_{12}} \quad \text{(K-App)}$$

$$\frac{\Gamma \vdash T_1 :: * \qquad \Gamma \vdash T_2 :: *}{\Gamma \vdash T_1 \to T_2 :: *} \quad \text{(K-Arrow)}$$

$$\frac{\Gamma, X{::}K_1 \vdash T_2 :: *}{\Gamma \vdash \forall X{::}K_1.T_2 :: *} \quad \text{(K-All)}$$

型等価関係 $\boxed{S \equiv T}$

$$T \equiv T \quad \text{(Q-Refl)}$$

$$\frac{T \equiv S}{S \equiv T} \quad \text{(Q-Symm)}$$

$$\frac{S \equiv U \qquad U \equiv T}{S \equiv T} \quad \text{(Q-Trans)}$$

$$\frac{S_1 \equiv T_1 \qquad S_2 \equiv T_2}{S_1 \to S_2 \equiv T_1 \to T_2} \quad \text{(Q-Arrow)}$$

$$\frac{S_2 \equiv T_2}{\forall X{::}K_1.S_2 \equiv \forall X{::}K_1.T_2} \quad \text{(Q-All)}$$

$$\frac{S_2 \equiv T_2}{\lambda X{::}K_1.S_2 \equiv \lambda X{::}K_1.T_2} \quad \text{(Q-Abs)}$$

$$\frac{S_1 \equiv T_1 \qquad S_2 \equiv T_2}{S_1\ S_2 \equiv T_1\ T_2} \quad \text{(Q-App)}$$

$$(\lambda X{::}K_{11}.T_{12})\ T_2 \equiv [X \mapsto T_2]T_{12} \quad \text{(Q-AppAbs)}$$

型付け $\boxed{\Gamma \vdash t : T}$

$$\frac{x{:}T \in \Gamma}{\Gamma \vdash x : T} \quad \text{(T-Var)}$$

$$\frac{\Gamma \vdash T_1 :: * \qquad \Gamma, x{:}T_1 \vdash t_2 : T_2}{\Gamma \vdash \lambda x{:}T_1.t_2 : T_1 \to T_2} \quad \text{(T-Abs)}$$

$$\frac{\Gamma \vdash t_1 : T_{11} \to T_{12} \qquad \Gamma \vdash t_2 : T_{11}}{\Gamma \vdash t_1\ t_2 : T_{12}} \quad \text{(T-App)}$$

$$\frac{\Gamma, X{::}K_1 \vdash t_2 : T_2}{\Gamma \vdash \lambda X{::}K_1.t_2 : \forall X{::}K_1.T_2} \quad \text{(T-TAbs)}$$

$$\frac{\Gamma \vdash t_1 : \forall X{::}K_{11}.T_{12} \qquad \Gamma \vdash T_2 :: K_{11}}{\Gamma \vdash t_1\ [T_2] : [X \mapsto T_2]T_{12}} \quad \text{(T-TApp)}$$

$$\frac{\Gamma \vdash t : S \qquad S \equiv T \qquad \Gamma \vdash T :: *}{\Gamma \vdash t : T} \quad \text{(T-Eq)}$$

▲ 図 30-1.　高階多相ラムダ計算 (F_ω)、続き

$\forall X{::}*.T$ は $\forall X.T$ と、$\{\exists X{::}*,T\}$ は $\{\exists X,T\}$ と省略して書くことにする。これにより、System F の項はそのまま F_ω の項として読むことができる。

同様に、第 24 章の存在型の元々の形式化において、束縛を X から $X{::}K$ に一般化することで、存在型の高階版も得られる。この拡張を図 30-2 に要約する。

→ ∀ ∃ ⇒ F_ω（図 30-1）と図 24-1 の拡張

新しい構文形式

$$\begin{array}{llll} \mathsf{T} & ::= & \ldots & \text{型：} \\ & & \{\exists \mathsf{X}::\mathsf{K}, \mathsf{T}\} & \text{存在型} \end{array}$$

新しい評価規則 $\boxed{t \longrightarrow t'}$

$$\texttt{let}\ \{X,x\}=(\{{*}T_{11},v_{12}\}\ \texttt{as}\ T_1)\ \texttt{in}\ t_2 \longrightarrow [X \mapsto T_{11}][x \mapsto v_{12}]t_2 \quad \text{(E-UnpackPack)}$$

$$\frac{t_{12} \longrightarrow t'_{12}}{\{{*}T_{11},t_{12}\}\ \texttt{as}\ T_1 \longrightarrow \{{*}T_{11},t'_{12}\}\ \texttt{as}\ T_1} \quad \text{(E-Pack)}$$

新しいカインド付け規則 $\boxed{\Gamma \vdash T :: K}$

$$\frac{\Gamma, X::K_1 \vdash T_2 :: *}{\Gamma \vdash \{\exists X::K_1, T_2\} :: *} \quad \text{(K-Some)}$$

新しい型等価関係の規則 $\boxed{S \equiv T}$

$$\frac{S_2 \equiv T_2}{\{\exists X::K_1, S_2\} \equiv \{\exists X::K_1, T_2\}} \quad \text{(Q-Some)}$$

新しい型付け規則 $\boxed{\Gamma \vdash t : T}$

$$\frac{\Gamma \vdash t_2 : [X \mapsto U]T_2 \qquad \Gamma \vdash \{\exists X::K_1, T_2\} :: *}{\Gamma \vdash \{{*}U, t_2\}\ \texttt{as}\ \{\exists X::K_1, T_2\} : \{\exists X::K_1, T_2\}} \quad \text{(T-Pack)}$$

$$\frac{\Gamma \vdash t_1 : \{\exists X::K_{11}, T_{12}\} \qquad \Gamma, X::K_{11}, x:T_{12} \vdash t_2 : T_2}{\Gamma \vdash \texttt{let}\ \{X,x\}=t_1\ \texttt{in}\ t_2 : T_2} \quad \text{(T-Unpack)}$$

▲ 図 30-2. 高階存在型

30.2 例

型演算子の上を動く抽象を用いる発展的なプログラミング例は、第 32 章で扱う。ここでは、もっと小さい例を示す。

289 ページにおいて、存在型で抽象データ型を表現したことを思い出そう。そのときはカウンタなどの ADT を実装したが、今回は同じ方法で二つ組の ADT を実装したいとしよう。この ADT は、二つ組を構築する演算と分解する演算を提供すべきである。さらに、これらの演算を**多相的**にして、任意の型 S と T の要素からなる二つ組を扱えるようにしたい。つまり、提供する抽象型は真の型ではなく、抽象的な型**構築子**（型**演算子**）となる。この ADT も以前の ADT と同じ意味で抽象的であるべきである。つまり、任意の S と T に対して、`pair` 演算は S の要素と T の要素を受け取り `Pair S T` の要素を返すということ、`fst` と `snd` は `Pair S T` の要素を受け取りそれぞれ S と T の要素を返すということ、そしてこの抽象を用いる人はこれら二つの事実しか知らなくてよいということである。

以上の要求から、二つ組の ADT が世界に公開すべきシグネチャが決まる。

```
PairSig = {∃Pair::*⇒*⇒*,
           {pair: ∀X. ∀Y. X→Y→(Pair X Y),
            fst: ∀X. ∀Y. (Pair X Y)→X,
            snd: ∀X. ∀Y. (Pair X Y)→Y}};
```

つまり二つ組の実装が提供すべきなのは、型演算子 `Pair` と、上記の型を持つ多相関数 `pair`、`fst`、`snd` である。

この型を持つパッケージを構築する一つの方法が以下である。

```
  pairADT =
     {*λX. λY. ∀R. (X→Y→R) → R,
      {pair = λX. λY. λx:X. λy:Y.
                λR. λp:X→Y→R. p x y,
       fst = λX. λY. λp: ∀R. (X→Y→R) → R.
                p [X] (λx:X. λy:Y. x),
       snd = λX. λY. λp: ∀R. (X→Y→R) → R.
                p [Y] (λx:X. λy:Y. y)}}  as PairSig;
▸ pairADT : PairSig
```

隠蔽表現型は、以前 (276 ページ)、二つ組を表すために用いた演算子 λX. λY. ∀R. (X→Y→R) → R である。その本体の要素 pair、fst、snd はそれぞれ適切な多相関数である。

pairADT を定義したら、これまでと同様の方法でアンパッケージ化できる。

```
  let {Pair,pair}=pairADT
  in pair.fst [Nat] [Bool] (pair.pair [Nat] [Bool] 5 true);
▸ 5 : Nat
```

30.3 性質

では、F_ω の基本的な性質（特に、定番の保存定理と進行定理）の証明に移ろう。この証明のアイディアは以前説明したものに似ているが、やや大きく複雑になった体系を扱う分、慎重に話を進める必要がある。特に、型等価関係の構造の分析が必要である。証明を短くするため、全称型だけを持つ F_ω、つまり図 30-1 で定義する体系を扱う。議論を拡張して存在型にも対応させるのは単純である。

基本的な性質

幾つかの簡単な性質から始める。これらは後で必要になる。

補題 30.3.1.［強化］　Γ, x:S, $\Delta \vdash$ T :: K ならば Γ, $\Delta \vdash$ T :: K　□

［証明］　カインド付け関係は項の変数束縛を参照しない。　□

以前は並べ替えと弱化を順に証明していたが、F_ω では趣向を変えて同時に証明しよう。

補題 30.3.2.［並べ替えと弱化］　Γ、Δ という文脈があり、ある Σ に対して、Δ は Γ, Σ の並べ替えで、正しい形式を持つものとする[†1]。つまり、Δ は Γ の拡張の並べ替えである。

(1)　$\Gamma \vdash$ T :: K ならば、$\Delta \vdash$ T :: K

(2)　$\Gamma \vdash$ t : T ならば、$\Delta \vdash$ t : T　□

［証明］　導出に関する単純な帰納法。　□

補題 30.3.3.［項の代入］　Γ, x:S, $\Delta \vdash$ t : T かつ $\Gamma \vdash$ s : S ならば、Γ, $\Delta \vdash$ [x ↦ s]t : T　□

［証明］　導出に関する帰納法による。（演習［★］。どこで補題 30.3.1 を用いるだろうか。補題 30.3.2 についてはどうだろうか。）　□

---

†1 （訳注）「正しい形式」は定義 30.3.18 で定義されている。

並行簡約 $\boxed{S \Rrightarrow T}$

$$T \Rrightarrow T \qquad \text{(QR-Refl)}$$

$$\frac{S_1 \Rrightarrow T_1 \qquad S_2 \Rrightarrow T_2}{S_1 \to S_2 \Rrightarrow T_1 \to T_2} \qquad \text{(QR-Arrow)}$$

$$\frac{S_2 \Rrightarrow T_2}{\forall X{::}K_1.S_2 \Rrightarrow \forall X{::}K_1.T_2} \qquad \text{(QR-All)}$$

$$\frac{S_2 \Rrightarrow T_2}{\lambda X{::}K_1.S_2 \Rrightarrow \lambda X{::}K_1.T_2} \qquad \text{(QR-Abs)}$$

$$\frac{S_1 \Rrightarrow T_1 \qquad S_2 \Rrightarrow T_2}{S_1\ S_2 \Rrightarrow T_1\ T_2} \qquad \text{(QR-App)}$$

$$\frac{S_{12} \Rrightarrow T_{12} \qquad S_2 \Rrightarrow T_2}{(\lambda X{::}K_{11}.S_{12})\ S_2 \Rrightarrow [X \mapsto T_2]T_{12}} \qquad \text{(QR-AppAbs)}$$

▲ 図 30-3. 型の並行簡約

補題 30.3.4.［型の代入］

(1) $\Gamma, Y{::}J, \Delta \vdash T :: K$ かつ $\Gamma \vdash S :: J$ ならば、$\Gamma, [Y \mapsto S]\Delta \vdash [Y \mapsto S]T :: K$
(2) $T \equiv U$ ならば、$[Y \mapsto S]T \equiv [Y \mapsto S]U$
(3) $\Gamma, Y{::}J, \Delta \vdash t : T$ かつ $\Gamma \vdash S :: J$ ならば、$\Gamma, [Y \mapsto S]\Delta \vdash [Y \mapsto S]t : [Y \mapsto S]T$ □

［証明］ 導出に関する単純な帰納法。K-TVar と T-Var の場合には弱化（補題 30.3.2）を用いる。Q-AppAbs の場合には、$[X \mapsto [Y \mapsto S]T_2]([Y \mapsto S]T_{12})$ と $[Y \mapsto S]([X \mapsto T_2]T_{12})$ が同じものであるという観察も必要である。 □

型等価性と簡約

F_ω における型付けの性質を証明するには、型等価関係に向きを与えたものを用いると便利である。この関係を**並行簡約**という（図 30-3 を参照)。型等価性との違いは、対称律と推移律が外されたことと、QR-AppAbs の規則が簡約基の一部分を同時に簡約することを許していることである。対称律を外すことで、簡約関係はより「計算的」な印象になっている。$(\lambda X{::}K_{11}.T_{12})\ T_2$ は $[X \mapsto T_2]T_{12}$ に簡約されるが、逆方向に簡約はできない。このように向きを与えることで、例えば後述する補題 30.3.12 の証明で、簡約関係の解析がやりやすくなっている。推移律を外すことと、ラムダ簡約基の簡約と同時に一部分の簡約を許すことは、細かい工夫である。この変更を行うと、簡約関係に関して、後述する補題 30.3.8 が主張している単一ステップでのダイヤモンド性が成り立つようになるのである。

並行簡約関係の重要な性質は、その推移的対称的閉包（$\Lleftarrow\!\!\Rrightarrow^*$ と書く）が型等価性と一致することである。

補題 30.3.5. $S \equiv T$ のとき、かつそのときに限り、$S \Lleftarrow\!\!\Rrightarrow^* T$ □

［証明］（$\Lleftarrow$）の方向は明らかである。（$\Rrightarrow$）の方向について唯一難しいのは、型等価性の導出が Q-Symm と Q-Trans のインスタンスを任意の箇所で用いるかもしれないのに対し、$\Lleftarrow\!\!\Rrightarrow^*$ 関係の定義は対称律と推移律を最も外側でしか用いることができないという点である。これに対処するには、以下の観察を用いればよい。$S \equiv T$ の任意の導出から、推移律を含まない導出の列 $S = S_0 \equiv S_1 \equiv S_2 \equiv \cdots \equiv S_n = T$ を推移律でつないだもの（ただし、各部分導出 $S_i \equiv S_{i+1}$ におい

て、Q-SYMM は一番最後の規則としてのみ用いられるか、または全く用いられない）へ変換できる。

□

さらに、並行簡約が**合流性**を持つことは容易にわかる。次の数個の補題で示す。（合流性はしばしば **Church-Rosser 性**といわれる。）

補題 30.3.6. $S \Rrightarrow S'$ ならば、任意の型 T に対して $[Y \mapsto S]T \Rrightarrow [Y \mapsto S']T$ □

［証明］ T の構造に関する帰納法による。 □

補題 30.3.7. $S \Rrightarrow S'$ かつ $T \Rrightarrow T'$ ならば、$[Y \mapsto S]T \Rrightarrow [Y \mapsto S']T'$ □

［証明］ $T \Rrightarrow T'$ の導出に関する帰納法による。QR-REFL の場合には補題 30.3.6 を用いる。QR-ABS、QR-APP、QR-ARROW、QR-ALL の場合は単純に帰納法の仮定を用いて進める。QR-APPABS の場合は、$T = (\lambda X{::}K_{11}.T_{12})\ T_2$ かつ $T' = [X \mapsto T_2']T_{12}'$ で、ただし $T_{12} \Rrightarrow T_{12}'$ かつ $T_2 \Rrightarrow T_2'$ である。帰納法の仮定により、$[Y \mapsto S]T_{12} \Rrightarrow [Y \mapsto S']T_{12}'$ かつ $[Y \mapsto S]T_2 \Rrightarrow [Y \mapsto S']T_2'$。QR-APPABS を適用して、$(\lambda X{::}K_{11}.[Y \mapsto S]T_{12})\ [Y \mapsto S]T_2 \Rrightarrow [X \mapsto [Y \mapsto S']T_2']([Y \mapsto S']T_{12}')$ を得る。つまり、$[Y \mapsto S]((\lambda X{::}K_{11}.T_{12})\ T_2) \Rrightarrow [Y \mapsto S']([X \mapsto T_2']T_{12}')$。 □

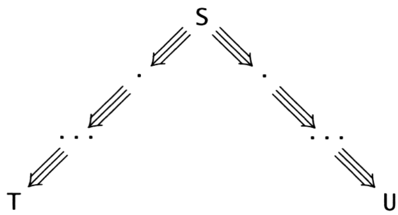
補題 30.3.8.［簡約の単一ステップでのダイヤモンド性］ $S \Rrightarrow T$ かつ $S \Rrightarrow U$ ならば、ある型 V が存在して $T \Rrightarrow V$ かつ $U \Rrightarrow V$ □

［証明］ 演習［推奨, ★★★］。 □

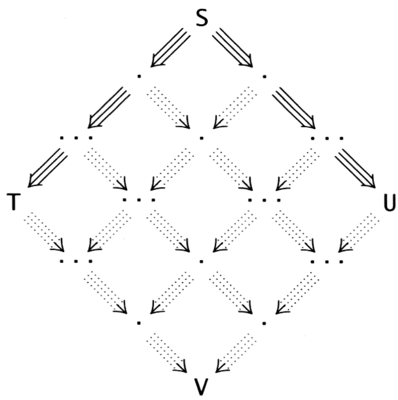
補題 30.3.9.［合流性］ $S \Rrightarrow^* T$ かつ $S \Rrightarrow^* U$ ならば、ある型 V が存在して $T \Rrightarrow^* V$ かつ $U \Rrightarrow^* V$

□

［証明］ S から T への簡約の個々のステップと、S から U へのステップを可視化する。



すると、補題 30.3.8 を繰り返し用いて図の内側をタイルのように埋め尽くすことができる。



これにより、大きなダイヤモンドが得られる。このダイヤモンドの下側の辺が、求める簡約の列である。 □

命題 30.3.10. $\mathsf{S} \Leftrightarrow^* \mathsf{T}$ ならば、ある U が存在して $\mathsf{S} \Rrightarrow^* \mathsf{U}$ かつ $\mathsf{T} \Rrightarrow^* \mathsf{U}$ □

［証明］ 演習［★★］。 □

ここから、等価性と簡約に関する極めて重要な観察を得られる。もし二つの型が等価ならば、共通の簡約結果を持つということである。ここから、この先で逆転の性質を証明するために必要な構造が得られる。

系 30.3.11. $\mathsf{S} \equiv \mathsf{T}$ ならば、ある U が存在して $\mathsf{S} \Rrightarrow^* \mathsf{U}$ かつ $\mathsf{T} \Rrightarrow^* \mathsf{U}$ □

保存

これで、簡約の前後で型が保存されることの証明本体に必要な準備はほぼ整った。もう一つだけ必要なのは、結論がある形を持つ型付け導出が与えられたときの、その部分導出の形に関する、定番の**逆転補題**である。そしてこの補題は、並行簡約に関する単純な観察に依存する。

補題 30.3.12.［簡約の下での形の保存］

(1) $\mathsf{S_1}\to\mathsf{S_2} \Rrightarrow^* \mathsf{T}$ ならば、$\mathsf{T} = \mathsf{T_1}\to\mathsf{T_2}$ であり、このとき $\mathsf{S_1} \Rrightarrow^* \mathsf{T_1}$ かつ $\mathsf{S_2} \Rrightarrow^* \mathsf{T_2}$

(2) $\forall\mathsf{X}::\mathsf{K_1}.\mathsf{S_2} \Rrightarrow^* \mathsf{T}$ ならば、$\mathsf{T} = \forall\mathsf{X}::\mathsf{K_1}.\mathsf{T_2}$ であり、このとき $\mathsf{S_2} \Rrightarrow^* \mathsf{T_2}$ □

［証明］ 単純な帰納法。 □

補題 30.3.13.［逆転］

(1) $\Gamma \vdash \lambda\mathsf{x}:\mathsf{S_1}.\ \mathsf{s_2} : \mathsf{T_1}\to\mathsf{T_2}$ ならば、$\mathsf{T_1} \equiv \mathsf{S_1}$ かつ $\Gamma, \mathsf{x}:\mathsf{S_1} \vdash \mathsf{s_2} : \mathsf{T_2}$。さらに $\Gamma \vdash \mathsf{S_1} :: *$

(2) $\Gamma \vdash \lambda\mathsf{X}::\mathsf{J_1}.\ \mathsf{s_2} : \forall\mathsf{X}::\mathsf{K_1}.\mathsf{T_2}$ ならば、$\mathsf{J_1} = \mathsf{K_1}$ かつ $\Gamma, \mathsf{X}::\mathsf{J_1} \vdash \mathsf{s_2} : \mathsf{T_2}$ □

［証明］ (1) については、若干一般化して、「$\Gamma \vdash \lambda\mathsf{x}:\mathsf{S_1}.\mathsf{s_2} : \mathsf{S}$ かつ $\mathsf{S} \equiv \mathsf{T_1}\to\mathsf{T_2}$ ならば、$\mathsf{T_1} \equiv \mathsf{S_1}$ かつ $\Gamma, \mathsf{x}:\mathsf{S_1} \vdash \mathsf{s_2} : \mathsf{T_2}$ である」という言明を帰納法によって証明する。帰納法のステップ（規則 T-EQ）は単純である。興味深いのは、帰納法の基底ケース（規則 T-ABS）のほうである。この場合、S は $\mathsf{S_1}\to\mathsf{S_2}$ の形を持ち、$\Gamma, \mathsf{x}:\mathsf{S_1} \vdash \mathsf{s_2} : \mathsf{S_2}$。系 30.3.11 と補題 30.3.12 (1) から $\mathsf{T_1} \equiv \mathsf{S_1}$ かつ $\mathsf{T_2} \equiv \mathsf{S_2}$ を得る。ここから、T-EQ より $\Gamma, \mathsf{x}:\mathsf{S_1} \vdash \mathsf{s_2} : \mathsf{T_2}$。さらに、T-ABS のもう一つの前提から、$\Gamma \vdash \mathsf{S_1} :: *$。(2) のほうも同様である。 □

定理 30.3.14.［保存］ $\Gamma \vdash \mathsf{t} : \mathsf{T}$ かつ $\mathsf{t} \longrightarrow \mathsf{t'}$ ならば、$\Gamma \vdash \mathsf{t'} : \mathsf{T}$ □

［証明］ 型付け導出に関する単純な帰納法。部分型のある単純型付きラムダ計算の保存（定理 15.3.5）の証明と同様である。

T-VAR の場合　$\mathsf{t} = \mathsf{x}$

起きえない（変数の評価規則は存在しない）。

T-ABS の場合　$\mathsf{t} = \lambda\mathsf{x}:\mathsf{T_1}.\mathsf{t_2}$

起きえない（t は既に値である）。

T-APP の場合　$\mathsf{t} = \mathsf{t_1}\ \mathsf{t_2}$　　$\Gamma \vdash \mathsf{t_1} : \mathsf{T_{11}}\to\mathsf{T_{12}}$　　$\Gamma \vdash \mathsf{t_2} : \mathsf{T_{11}}$　　$\mathsf{T} = \mathsf{T_{12}}$

図 30-1 より、$\mathsf{t} \longrightarrow \mathsf{t'}$ が導出可能な規則は三つあるとわかる。E-APP1、E-APP2、E-APPABS であ

る。最初の二つについては、単純に帰納法の仮定を用いて結論が得られる。三つ目についてはより興味深い。

E-AppAbs の場合　$t_1 = \lambda x{:}S_{11}.\ t_{12}$　　$t_2 = v_2$　　$t' = [x \mapsto v_2]t_{12}$
補題 30.3.13 (1) より、$T_{11} \equiv S_{11}$ かつ $\Gamma, x{:}S_{11} \vdash t_{12} : T_{12}$。T-Eq により、$\Gamma \vdash t_2 : S_{11}$。これと代入補題（補題 30.3.3）から、$\Gamma \vdash t' : T_{12}$ を得る。

T-TAbs の場合　$t = \lambda X{::}K_1.t_2$
起きえない（t は既に値である）。

T-TApp の場合　$t = t_1\ [T_2]$　　$\Gamma \vdash t_1 : \forall X{::}K_{11}.T_{12}$　　$\Gamma \vdash T_2 :: K_{11}$
$T = [X \mapsto T_2]T_{12}$
T-App の場合と同様である。項の代入補題（補題 30.3.3）の代わりに型の代入補題（補題 30.3.4）を用いる。

T-Eq の場合　$\Gamma \vdash t : S$　　$S \equiv T$　　$\Gamma \vdash T :: *$
帰納法の仮定により、$\Gamma \vdash t' : S$。T-Eq から、$\Gamma \vdash t' : T$。□

進行

次は進行定理を考える。ここでも、証明のために必要なものはほとんどそろっている。あと必要なのは、閉じた値の形に関する、いつもの標準形補題である。

補題 30.3.15.［標準形］

(1)　t が閉じた値で、$\vdash t : T_1 \to T_2$ ならば、t は抽象である。
(2)　t が閉じた値で、$\vdash t : \forall X{::}K_1.T_2$ ならば、t は型抽象である。□

［証明］(1) も (2) も議論は同様なので、(1) だけ示す。値には二つの形式しかないから、t が値で抽象でなければ、型抽象でなければならない。よって、（矛盾を導くために）t が型抽象であると仮定する。$\vdash t : T_1 \to T_2$ の型付け導出は、T-TAbs を使った後、T-Eq の使用が一つ以上並んで終わりとならなければならない。つまり、以下の形でなければならない（カインド付けの前提は省略している）。

$$\dfrac{\dfrac{\vdots}{\vdash t : \forall X{::}K_{11}.S_{12}}\,\text{(T-TAbs)} \qquad \forall X{::}K_{11}.S_{12} \equiv U_1}{\vdash t : U_1}\,\text{(T-Eq)}$$

$$\vdots$$

$$\dfrac{\dfrac{\vdash t : U_{n-1} \qquad U_{n-1} \equiv U_n}{\vdash t : U_n}\,\text{(T-Eq)} \qquad U_n \equiv T_1 \to T_2}{\vdash t : T_1 \to T_2}\,\text{(T-Eq)}$$

型等価性は推移的なので、すべての等価性の使用を一つに畳み込むことができ、その結果 $\forall X{::}K_{11}.S_{12} \equiv T_1 \to T_2$ といえる。さて、系 30.3.11 より、ある型 U が存在して $\forall X{::}K_{11}.S_{12} \Rrightarrow^* U$ かつ $T_1 \to T_2 \Rrightarrow^* U$ でなければならない。補題 30.3.12 により、そのような U はその最も外側の構築子として量化子と矢印の両方を持たなければならず、矛盾。□

定理 30.3.16.［進行］　t を、閉じた、正しく型付けされた項とする（つまり、ある T に対して $\vdash t : T$）。このとき、t は値であるか、さもなくばある t′ が存在して $t \longrightarrow t'$ □

[証明] 型付け導出に関する帰納法による。T-VAR の場合は、t が閉じていることから起きえない。T-ABS と T-TABS の場合は、抽象が値であることから直ちに証明できる。T-EQ の場合は帰納法の仮定により直接導かれる。残る適用と型適用の場合は、より興味深い。型適用の場合だけ証明する。適用のほうも同様である。

T-TAPP の場合 $t = t_1\ [T_2]$ $\quad \vdash t_1 : \forall X{::}K_{11}.T_{12}$ $\quad \vdash T_2 :: K_{11}$

帰納法の仮定により、t_1 は値であるか、さもなくば評価を進めることができる。t_1 が評価のステップを持つならば、規則 E-TAPP を t に適用できる。t_1 が値ならば、標準形補題 (補題 30.3.15) により、t_1 は型抽象であるとわかる。よって規則 E-TAPPTABS を t に適用できる。 □

演習 30.3.17. [推奨, ★★] 型等価関係に次の妙な規則を追加するとしよう。

$$T \to T \equiv \forall X{::}*.T$$

この体系の基本的な性質はどちらか壊れてしまうだろうか。一方、次の規則を追加するとする。

$$S \to T \equiv T \to S$$

この場合はどうだろうか。 □

カインド付け

図 30-1 の F_ω の定義では、項に対して規則を用いて導出できる型が正しくカインド付けされていることを保証するのに苦労していた。特に T-ABS は、ラムダ抽象の型注釈を文脈に追加する前に、それが正しい形式をしていることを検査している。また T-EQ は、t が持つことになる型 T がカインド $*$ を持つことを検査している。これらの検査は正しい形式を持つことを保証しているのだが、その正確な意味は以下の命題によって与えられる。

定義 30.3.18. 文脈 Γ は次のいずれかの条件を満たすとき、**正しい形式を持つ**という。(1) Γ は空である。(2) $\Gamma = \Gamma_1, x{:}T$、ただし Γ_1 は正しい形式を持ち、$\Gamma_1 \vdash T :: *$ である。(3) $\Gamma = \Gamma_1, X{::}K$、ただし Γ_1 は正しい形式を持つ。 □

命題 30.3.19. $\Gamma \vdash t : T$ かつ Γ が正しい形式を持つならば、$\Gamma \vdash T :: *$ □

[証明] 定型的な帰納法。T-TAPP の場合には補題 30.3.4 (1) を用いる。 □

決定可能性

紙面の都合により、F_ω の決定可能性の完全な証明 (型検査アルゴリズムと、その健全性、完全性、停止性の証明) は本書に含めないが、必要なアイディアのほとんどは第 28 章の System $F_{<:}$ の最小型付けアルゴリズムで既に触れている。

カインド付け関係は決定可能 (規則が構文主導なので) という観察から始まる。カインド付けは本質的に単純型付きラムダ計算の「1 レベル上」の複製だと知っているので、驚くことではない。このことから、型付け規則において正しくカインド付けされているかどうかの検査は実効的に実装可能だとわかる。

次に、構文主導でない規則、すなわち T-EQ を型付け関係から取り除く。$F_{<:}$ で T-SUB を取り除いたのと同じである。そして失われた T-EQ 規則が本質的に用いられていたところを埋め合わせるために、他の規則で前提を一般化しなければならない箇所を調べる。T-EQ 規則が欠かせない箇所が二つあることがわかる。

(1) 規則 T-App と T-TApp の一つ目の前提において、左辺の部分式 t_1 の型を書き換えて、矢印または量化子が外側に現れるように、T-Eq を用いる必要がある可能性がある。(例えば、もし文脈が変数 x を型 (λX.X→X)Nat に関連付けているならば、適用 x 5 は型 Nat を持つ。このようになるのは、x の型を Nat→Nat に書き換え可能だからこそである。)
これと同じ効果を得るために、28.1 節の露出関係のようなものを導入する。ここでは、t_1 の最小型が矢印か量化子になるまで**昇格**させるのではなく、必要に応じて**簡約**する。例えば、非自明な簡約が不可能になるまで図 30-3 の規則を繰り返し適用するということである[†2]。
この処理が停止することを確信するには、この簡約規則が正規化可能であることを証明する必要がある。もちろん、正しくカインド付けされていない項においては、簡約は正規化可能で**ない**。なぜなら F_ω の型の構文には、omega (49 ページ) のように発散する項を表現するために必要なプリミティブがすべてそろっているからである。幸いなことに、命題 30.3.19 より、正しい形式を持つ文脈から始める限り (そして、文脈に追加する注釈がどれも正しくカインド付けされていることを保証するよう、適切にカインド検査を行う限り)、正しくカインド付けされた項だけを扱えばよい。そして、そのような項に対して、簡約が常に一意の正規形に至ることを証明可能である (例えば第 12 章の技法を適用すればよい)。

(2) T-App の二つ目の前提において、t_2 に対して計算された型 T_2 を、t_1 の矢印型の左辺 T_{11} と一致させるため、等価性を用いる必要がある可能性がある。したがってこの規則をアルゴリズム的にしたものは、T_2 と T_{11} の等価性**検査**を含む。この検査を実装するには、例えば、T_2 と T_{11} の両方をそれぞれ正規形になるまで簡約して、それらが (束縛変数名の違いを除いて) 一致するかどうかをテストすればよい。

演習 30.3.20. [★★★★]　これらのアイディアに基づいて F_ω の検査器を実装せよ。purefsub の検査器を出発点として用いよ。 □

30.4 制限された F_ω

直観的には、$\lambda_\rightarrow$ と System F が F_ω に含まれるのは明らかである。この直観を正確にするには、System F_1、F_2、F_3、…という階層構造で、極限が F_ω であるようなものを定義するとよい。

定義 30.4.1. System F_1 においては、カインドは * だけで、型の量化 (∀) や抽象 (λ) は許さない。他の体系は、**レベル *i* のカインド** (以下に定義) の階層構造を用いて定義できる。

$$\begin{array}{lcl} \mathcal{K}_1 & = & \emptyset \\ \mathcal{K}_{i+1} & = & \{*\} \cup \{J \Rightarrow K \mid J \in \mathcal{K}_i \text{ かつ } K \in \mathcal{K}_{i+1}\} \\ \mathcal{K}_\omega & = & \bigcup_{1 \leq i} \mathcal{K}_i \end{array}$$

System F_2 においては、やはりカインドは * だけで、型レベルのラムダ抽象は持たないが、真の型 (カインドは *) に対する量化はできる。F_3 では、型演算子に対する量化が可能である (つまり、∀X::K.T の形の型式を書ける。ただし $K \in \mathcal{K}_3$) ことに加え、真の型に対する抽象を導入する (つまり λX::*.T の形の型式を考えられる。そのカインドは *⇒* のようになる)。F_{i+1} においては一般に、$\mathcal{K}_{i+1}$ のカインドを持つ型の量化と、$\mathcal{K}_i$ のカインドを持つ型の抽象が可能である。 □

[†2] 実際には、F_ω の型検査器のほとんどは、**弱頭部簡約**といわれる、より消極的な簡約を用いている。弱頭部簡約では、最左・最外の簡約基だけを簡約し、何かの具象構築子 (適用以外の何か) が型の先頭にきたら停止する。

F_1 はただの単純型付きラムダ計算 $\lambda_\to$ である。その定義は図 9-1 より一見複雑である。これはカインド付けや型等価性の定義があるためであるが、ここではいずれも自明である。というのは、構文的に正しい形式を持つ型はカインド * として正しくカインド付けされているし、型 T と等価な型は T 自身だけだからである。F_2 は System F である。System F がしばしば**二階ラムダ計算**と呼ばれるのは、この階層構造の位置に由来する。F_3 はカインド付けや型等価関係が非自明になる最初の体系である。

興味深いことに、本書のプログラムはすべて F_3 に含まれる（厳密に言うと、第 32 章の型演算子 `Object` は F_4 である。なぜなら、その引数がカインド (*⇒*)⇒* の型演算子であるからである。しかし、第 29 章以前では `Pair` を略記として扱っていたように、`Object` を計算体系の正式な式としてではなく、メタ言語の略記の機構で扱うことにすればよい。なぜなら、`Object` を用いる例で、このカインドの型を**量化**する必要があるものはないからである）。ただし、本書の言語を完全な F_ω でなく F_3 に制限しても、実装の難しさやメタ理論の複雑さの観点で実質的に大きく簡略化されるわけではない。なぜなら、型演算子抽象と型等価性という重要な仕組みは既にこのレベルに含まれているからである。

演習 30.4.2. [★★★★ ↛]　F_3 では書けないが F_4 なら書けるプログラムで、有用なものは存在するだろうか。 □

30.5 発展：依存型

本書の大部分では、さまざまなものを抽象化する機構の形式化に取り組んできた。単純型付きラムダ計算では、項の中の部分項を抽象化する操作を形式化した。これにより、後で異なる項に適用して具体化できる関数を得られた。System F では、項の中の**型**を抽象化する操作を考えた。これにより、さまざまな型に適用して具体化できる項を得られた。λ_ω では、単純型付きラムダ計算の仕組みを「1 レベル上げて」反復した。この仕組みは、型を受け取り、その部分式を抽象化する。これにより、後で異なる型に適用することで具体化できる型演算子を得た。

以上すべての抽象化を考える上では、他の式で**インデックス付け**された式の**族**という観点が便利である。通常のラムダ抽象 $\lambda x{:}T_1.t_2$ は、項 s でインデックス付けされた項 $[x \mapsto s]t_2$ の族である。同様に、型抽象 $\lambda X{::}K_1.t_2$ は、型でインデックス付けされた項の族であり、型演算子は型でインデックス付けされた型の族である。

$\lambda x{:}T_1.t_2$	項でインデックス付けされた項の族
$\lambda X{::}K_1.t_2$	型でインデックス付けされた項の族
$\lambda X{::}K_1.T_2$	型でインデックス付けされた型の族

このリストを見ると、明らかにまだ考えていない可能性がある。すなわち、**項**でインデックス付けされた**型**である。そのような抽象化も、**依存型**という名前で広く研究されてきた。

依存型を用いると、プログラムの振る舞いをより正確に記述できる。その正確さは、今まで見てきた他の型付け機能を遥かに超える。単純な例として、`FloatList` という組み込みの型と、それに対する定番の操作があるとしよう。

```
nil    :  FloatList
cons   :  Float → FloatList → FloatList
hd     :  FloatList → Float
tl     :  FloatList → FloatList
isnil  :  FloatList → Bool
```

依存型のある言語では、この単純な型 FloatList を詳細化して型の族 FloatList n、すなわち n 個の要素を持つリストの型とすることができる。

この詳細化を生かすため、基本的なリスト演算の型も細かくしていく。まず、定数 nil に型 FloatList 0 を与える。他の操作にもより正確な型を与えるために、関数型の記法を詳細化して、引数と結果の型との間の**依存性**を表現できるようにする必要がある。例えば、cons の型は大雑把に言って「Float と長さ n のリストを受け取り、長さ n+1 のリストを返す関数」とならなければならない。n を第一引数として与えることで n の束縛を明示すれば、その記述は「数 n、Float、長さ n のリストを受け取り、長さ succ n のリストを返す関数」となる。つまり、この型で捉える必要があるのは、第一引数の**値**（n）と、第三引数および返り値の**型**（FloatList n と FloatList (succ n)）との間にある依存関係である。このため、Nat→... の代わりに Πn:Nat. ... と書いて、第一引数に名前を束縛する。すると cons とその他のリスト操作の型は次のようになる。

```
nil   :  FloatList 0
cons  :  Πn:Nat. Float → FloatList n → FloatList (succ n)
hd    :  Πn:Nat. FloatList (succ n) → Float
tl    :  Πn:Nat. FloatList (succ n) → FloatList n
```

nil、cons、tl の型からは、その返り値が要素を幾つ持つか正確にわかる。また、hd と tl が引数として空でないリストを要求していることもわかる。さらに、isnil はもはや不要であることに注意されたい。FloatList n の要素が nil かどうか判断するには、n が 0 かどうかを検査するだけで十分である。

Πx:T_1.T_2 という形を持つ**依存関数型**は、矢印型 $T_1 \to T_2$ をより正確にした形式である。ここで関数の引数を表す変数 x を束縛することで、返り値の型 T_2 の中でその引数を参照できるようにする。縮退している場合、つまり T_2 が x に言及していない場合には、Πx:T_1.T_2 を $T_1 \to T_2$ と書く。

もちろん、依存関数型を持つ新しい項を定義することもできる。例えば以下の関数を考える。

```
consthree = λn:Nat. λf:Float. λl:FloatList n.
              cons (succ(succ n)) f
                (cons (succ n) f
                   (cons n f l));
```

この関数は第二引数（f）を三回複製し、第三引数（l）の先頭に追加する。この関数の型は以下のようになる。

```
Πn:Nat. Float → FloatList n → FloatList (succ(succ(succ n)))
```

cons を三回呼び出しているが、それぞれ第一引数が異なることに注意されたい。これはリスト引数の長さが異なることに対応している。

計算機科学や論理学において、依存型の文献は豊富にある。出発点としてよいのは、Smith, Nordström, and Petersson [436]、Thompson [460]、Luo [290]、Hofmann [213] である。

演習 30.5.1. [★★]　例を簡単にするためリストの要素の型を Float に固定したが、通常の型演算

子を用いることで、この型を任意の型Tのリストに一般化できる。そのやり方を示せ。 □

同様の方針に則して、同じように詳細化した型を持つ高水準のリスト操作関数を定義できる。例えば、以下の型を持つようなソート関数を書ける。

```
sort : Πn:Nat. FloatList n → FloatList n
```

この型からは、このソート関数が返すリストは引数と同じ長さだとわかる。実際、関連する型の族をもっと詳細化することで、返り値のリストが常にソートされているとわかる型を持つ `sort` 関数すら書ける。関数 `sort` が実際にこの型に属すことを検査することは、要するに関数がその仕様を満たしていることを**証明**するのと同じなのである。

このような例を見ると、プログラムが**構成により正しい**という、魅力的な世界像を想像してしまう。この世界では、プログラムの振る舞いについて知りたいことはすべて型が教えてくれる。型検査器が "ok" と言えばプログラムが期待通りに振る舞うことに絶対の自信を持てる。この考え方に関連するのは、仕様を満たすことの証明から「計算的な内容を取り出す」ということによるプログラミングのアイディアである。このアイディアのための重要な観察は次のようなものである。「すべての x に対し、P であるような y が存在する」という形の定理の構成的証明は、x から y へ写す関数と、この関数が性質 P を満たすことの証拠（これは計算にとって本質的なものではなく、型検査器だけが興味を持つものである）の組とみなせる。これらのアイディアは、NuPRL（Constable et al. [110]）、LEGO（Luo and Pollack [291]; Pollack [390]）、Coq（Paulin-Mohring [352]）プロジェクトなどの研究者たちが追求している。

残念ながら、依存型の強力さは諸刃の剣である。型の検査と任意の定理の証明との区別をぼやけさせたところで、定理の証明が魔法のように簡単になるわけではない。むしろ、型検査が計算的に扱えなくなってしまう。機械的証明支援系に取り組む数学者は、定理を入力し、ボタンを押して、Yes か No が返ってくるのをただ座って待てばよいわけではない。ツールが証明を構成・検証できるように誘導するため、**証明のスクリプト**と**タクティック**（証明戦略）を書くことに莫大な労力を費やす。構成による正しさの考え方が極限に達すると、プログラマは型検査器を誘導するために、プログラムに対してヒントや期待を注釈することに同じくらいの労力を費やすことになる。一部の極めて重要なプログラミング業務ならこの努力に見合うかもしれないが、日常的なプログラミングでは間違いなくコストが高すぎる。

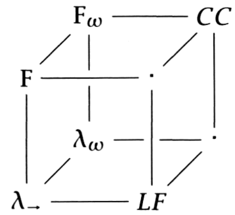
それにもかかわらず、実用的なプログラミング言語の設計で依存型を用いる試みは幾つかある。Russell（Donahue and Demers [138]; Hook [217]）、Cayenne（Augustsson [30]）、Dependent ML（Xi and Pfenning [502; 503]）、Dependently Typed Assembly Language（Xi and Harper [501]）、**シェイプ型**（Jay and Sekanina [237]）などである。これらの言語の最近の興味は、依存型の強力さをさまざまな方法で制限し、もっと自動化された型検査器が作れる扱いやすい体系を得ることにある。例えば Xi らの言語では、配列アクセスの実行時境界検査を静的に除去するためだけに依存型を用いている。これらの言語の型検査で発生する「定理証明」問題は、線形制約系にすぎない。この問題に対してはよい自動的手続きが存在する。

依存型が伝統的にプログラミング言語に影響を与えてきた分野としては、モジュールをまたがる依存性の間の**シェアリング**を追跡するための仕組みを持つ**モジュールシステム**の設計がある。この分野の著名な成果は、Pebble（Burstall and Lampson [68]）、MacQueen [296]、Mitchell and Harper [323]、Harper et al. [197]、Harper and Stone [199] などである。この分野の最近の論文は、技術的な工夫として**一元カインド**を適用している。これによると、モジュールの依存性は型ではなくカインドレベルで追跡される（例えば Stone and Harper [445]、Crary [121] を参照せよ。Hayashi [201] や Aspinall [26] も参照されたい）。

依存型と部分型の組み合わせをはじめて検討したのは Cardelli [77] である。さらに発展、一般化させたのは、Aspinall [26]、Pfenning [358]、Aspinall and Compagnoni [27]、Chen and Longo [97]、および Zwanenburg [507] である。

計算機科学の分野で依存型を適用した他の重要な事例は、証明支援系や自動定理証明器の構築である。特に、依存型を持つ単純型の体系はしばしば **logical framework** と呼ばれる。その中で最も有名なのは、純粋なラムダ計算に依存型を足した **LF**（Harper, Honsell, and Plotkin [195]）である。LF とその派生版、特に **Calculus of Constructions**（Coquand and Huet [118]; Luo [290]）は、定理証明環境の血統（AutoMath（de Bruijn [135]）、NuPRL（Constable et al. [110]）、LEGO（Luo and Pollack [291]; Pollack [390]）、Coq（Barras et al. [41]）、ALF（Magnusson and Nordström [297]）、ELF（Pfenning [359]）など）の基礎となっている。Pfenning [360] がこの分野についてより詳しくサーベイしている。

本節の冒頭で議論した抽象の四つの形式は、以下の図できれいに要約される。この図は **Barendregt キューブ**と呼ばれる[†3]。



このキューブの体系はすべて通常の項の抽象を持つ。上面は多相性（型でインデックス付けされた項の族）を持つ体系を表現している。背面は型演算子のある体系、右の面は依存型のある体系である。右奥の角は Calculus of Constructions で、四つの抽象の形式すべてを含むものである。上で言及してきた他の角としては、LF、つまり依存型のある単純型付きラムダ計算がある。Barendregt キューブのすべての体系は、その他多くの体系も含めてであるが、**純粋型システム**（Terlouw [458]; Berardi [43]; Barendregt [39; 40]; Jutting, McKinna, and Pollack [245]; McKinna and Pollack [305]; Pollack [390]）という一般的な仕組みのインスタンスとして表現できる。

[†3] Barendregt [39] では**ラムダキューブ**と呼ばれている。

第 31 章

高階部分型付け

本章で扱う体系は純粋な $F^{\omega}_{<:}$（図 31-1）である。対応する実装は fomsub である（実装 fullfomsub は存在型などのさまざまな拡張を含んでいる）。

本書で取り扱う最後の型システムは $F^{\omega}_{<:}$（「エフ・オメガ・サブ」）と呼ばれ、これも本書のこれまでの部分で個別に学んだ機能を組み合わせたものである。今回組み合わせるのは、型演算子と部分型付けである。この型システムは有界量化を持つ二階の多相ラムダ計算 System $F_{<:}$ を型演算子で拡張したものとみなせる。最も興味深い新機能は、部分型関係をカインド * 上の関係から、高階のカインドに属する型の関係へと拡張することである。

$F^{\omega}_{<:}$ には幾つかの異なるバージョンが提案されており、それらは表現力やメタ理論の複雑さの点で異なっている。本書で取り上げるのは、Pierce and Steffen [371] によるバージョンに非常に近く、最も単純なものの一つである。本書ではこの型システムの性質の証明は一切行わない。それらに興味のある読者は同様の型システムを扱っている Pierce and Steffen [371] や Compagnoni [107]、Abadi and Cardelli [4] を参照してほしい。（第 28 章の複雑さを、30.3 節の複雑さで倍加することを考えれば、これらの証明にどれだけのスペースが必要になるかわかるだろう。）

$F^{\omega}_{<:}$ について議論する主な理由は、オブジェクト指向プログラミングに関する最後の事例（第 32 章）の基礎となるからである。そこでの例は、$F^{\omega}_{<:}$ の定義の深遠な側面は用いない。単に与えられた型演算子の部分型の上を動くような有界量化子を書けさえすればよいからである。そのため、初読の際には本章をざっと読み、後で疑問が出たときに戻ってきてもよい。

31.1 直観

部分型付けおよび有界量化と、型演算子との間には相互作用があるため、それらを組み合わせた体系の形式化では、幾つかの設計上の問題が生じる。この体系の定義へと進む前に、それらについて簡潔に論じておこう。

最初の問いは、部分型付けが存在するときに、$\lambda X{::}K_1.T_2$ のような型演算子は、$\lambda X{<:}T_1.T_2$ という形の**有界型演算子**へと一般化されるべきだろうか、という問いである。本章では規則性よりも単純さを優先し、有界量化と**有界でない**型演算子を持つ体系を定義する。

次の問題は、部分型関係をどのように拡張し型演算子を含むようにするかである。これには幾つかの選択肢が存在する。本章で選択する、最も単純な方法は、元の部分型関係を、**各引数での**関係として型演算子の上の部分型関係に持ち上げるというものである。ラムダ抽象に関しては、$\lambda X.S$ が $\lambda X.T$ の部分型であるのは、両者に任意の引数 U を与えたときに、それらの結果が部分型関係にあるときである。例えば、$\lambda X.Top{\to}X$ は $\lambda X.X{\to}Top$ の部分型である。これは任意の U について、$Top{\to}U$ は $U{\to}Top$ の部分型であるためである。言い換えれば、$\lambda X.S$ が $\lambda X.T$ の部分型であるのは、X を**抽象的**なままに保つ場合（すなわち、X の部分型や上位型に関する仮定を全く置かない場合）に、S が T の部分型であるときであるといえる。この後者の見方から直接、以下の規則が得られる。

$$\frac{\Gamma, X \vdash S <: T}{\Gamma \vdash \lambda X.S <: \lambda X.T} \qquad \text{(S-ABS)}$$

逆に、F と G が型演算子で $F <: G$ が成り立つならば、$F\ U <: G\ U$ が成り立つ。

$$\frac{\Gamma \vdash F <: G}{\Gamma \vdash F\,U <: G\,U} \tag{S-App}$$

ただし、この規則は F と G を**同一の**引数 U に適用した場合にしか適用できないことに注意されたい。F が各引数において G の部分型であるということからは、異なる引数に対して適用した場合のことは何もわからないのである。($F^{\omega}_{<:}$ のより複雑な変種の幾つかでは、異なる引数に適用した場合を考慮している。これらについては 31.4 節で言及する。)

型等価関係の意図から、もう一つの規則が生じる。$S \equiv T$ であれば、S と T は同じ要素を持つ。しかし、同じ要素を持つ型同士は互いに確実にお互いの部分型になっている。ここから、定義による等価性を基底ケースとして含む、もう一つの部分型規則が得られる。

$$\frac{\Gamma \vdash S :: K \qquad \Gamma \vdash T :: K \qquad S \equiv T}{\Gamma \vdash S <: T} \tag{S-Eq}$$

部分型付けをカインド $*$ からカインド $*{\Rightarrow}*$ に持ち上げたのと同様のことを繰り返して、より複雑なカインドへも部分型付けを持ち上げられる。例えば、P と Q がカインド $*{\Rightarrow}*{\Rightarrow}*$ の型演算子ならば、各 U に対して適用 P U が適用 Q U の、$*{\Rightarrow}*$ カインドにおける部分型であるとき、$P <: Q$ であると定義する。

この定義の有用な副作用は、高階カインドの部分型関係がすべて最大元を持つことである。$\mathsf{Top}[*] = \mathsf{Top}$ とおいて、

$$\mathsf{Top}[K_1{\Rightarrow}K_2] \quad \stackrel{\text{def}}{=} \quad \lambda X{::}K_1.\mathsf{Top}[K_2]$$

と定義すれば、単純な帰納法により、$\Gamma \vdash S <: \mathsf{Top}[K]$(ただし S はカインド K を持つとする)であることがわかる。以下の節での規則ではこの副作用を活用する。

通常の有界量化子から、**高階有界量化子**への一般化の過程は単純である。$F^{\omega}_{<:}$ は $F_{<:}$ の $\forall X{<:}T_1.T_2$ という形の有界量化子を受け継ぐ。高階のものへと(すなわち型演算子上の量化へと)一般化しても、この構文を変更する必要はない。ここでの T_1 が(型演算子を含め)任意の型式でよいということを、確認するだけである。F_ω から受け継ぐ有界でない高階量化子に関しては、最大の上界を持つ有界量化子の略記として扱える。すなわち、$\forall X{::}K_1.T_2$ は $\forall X{<:}\mathsf{Top}[K_1].T_2$ の略記として取り扱う。

最後に、$F^{\omega}_{<:}$ が、$F_{<:}$ から受け継いでいる問題として、S-All 規則は、より扱いやすいカーネル版か、より強力なフル版のいずれを用いるかという点がある。ここではカーネル版を用いる。フル版も意味論上は妥当であるものの、メタ理論的な性質は(フル $F_{<:}$ との類推から成り立つと期待されるものも含め)いまだ確立されていない。

31.2 定義

$F^{\omega}_{<:}$ を定義する規則を図 31-1 に列挙する。細かい点として、この体系は型変数の二種類の束縛(型演算子で用いられる $X{::}K$ および量化子で用いられる $X{<:}T$)を提供しているものの、文脈には後者しか用いることができないという点がある。K-Abs 規則および S-Abs 規則で、束縛子 $X{::}K$ をターンスタイルの右側から左側へと移動する際には、$X{<:}\mathsf{Top}[K]$ に変更する。もう一つの細かい点は、$F_{<:}$ の S-Refl および、F_ω の T-Eq が、$F^{\omega}_{<:}$ では省かれている点である。元の S-Refl のインスタンスは、S-Eq と Q-Refl の直接の帰結であり、一方 T-Eq は T-Sub と S-Eq から導出できる。

$\rightarrow \forall \Rightarrow$ <: Top　　F_ω（図 30-1）とカーネル $F_{<:}$（図 26-1）の拡張

構文

t	::=		項：
		x	変数
		λx:T.t	ラムダ抽象
		t t	関数適用
		λX<:T.t	型抽象
		t [T]	型適用
v	::=		値：
		λx:T.t	ラムダ抽象値
		λX<:T.t	型抽象値
T	::=		型：
		Top	最大の型
		X	型変数
		T→T	関数の型
		∀X<:T.T	全称型
		λX::K.T	演算子抽象
		T T	演算子適用
Γ	::=		文脈：
		∅	空の文脈
		Γ, x:T	項変数の束縛
		Γ, X<:T	型変数の束縛
K	::=		カインド：
		*	真の型のカインド
		K⇒K	演算子のカインド

評価　　$t \longrightarrow t'$

$$\frac{t_1 \longrightarrow t_1'}{t_1\ t_2 \longrightarrow t_1'\ t_2} \quad \text{(E-APP1)}$$

$$\frac{t_2 \longrightarrow t_2'}{v_1\ t_2 \longrightarrow v_1\ t_2'} \quad \text{(E-APP2)}$$

$$(\lambda x{:}T_{11}.t_{12})\ v_2 \longrightarrow [x \mapsto v_2]t_{12} \quad \text{(E-APPABS)}$$

$$\frac{t_1 \longrightarrow t_1'}{t_1\ [T_2] \longrightarrow t_1'\ [T_2]} \quad \text{(E-TAPP)}$$

$$(\lambda X{<:}T_{11}.t_{12})\ [T_2] \longrightarrow [X \mapsto T_2]t_{12} \quad \text{(E-TAPPTABS)}$$

カインド付け　　$\Gamma \vdash T :: K$

$$\Gamma \vdash \mathsf{Top} :: * \quad \text{(K-TOP)}$$

$$\frac{X{<:}T \in \Gamma \qquad \Gamma \vdash T :: K}{\Gamma \vdash X :: K} \quad \text{(K-TVAR)}$$

$$\frac{\Gamma, X{<:}\mathsf{Top}[K_1] \vdash T_2 :: K_2}{\Gamma \vdash \lambda X{::}K_1.T_2 :: K_1 \Rightarrow K_2} \quad \text{(K-ABS)}$$

$$\frac{\Gamma \vdash T_1 :: K_{11} \Rightarrow K_{12} \qquad \Gamma \vdash T_2 :: K_{11}}{\Gamma \vdash T_1\ T_2 :: K_{12}} \quad \text{(K-APP)}$$

$$\frac{\Gamma \vdash T_1 :: * \qquad \Gamma \vdash T_2 :: *}{\Gamma \vdash T_1 \rightarrow T_2 :: *} \quad \text{(K-ARROW)}$$

$$\frac{\Gamma, X{<:}T_1 \vdash T_2 :: *}{\Gamma \vdash \forall X{<:}T_1.T_2 :: *} \quad \text{(K-ALL)}$$

型等価関係　　$S \equiv T$

$$T \equiv T \quad \text{(Q-REFL)}$$

$$\frac{T \equiv S}{S \equiv T} \quad \text{(Q-SYMM)}$$

$$\frac{S \equiv U \qquad U \equiv T}{S \equiv T} \quad \text{(Q-TRANS)}$$

$$\frac{S_1 \equiv T_1 \qquad S_2 \equiv T_2}{S_1 \rightarrow S_2 \equiv T_1 \rightarrow T_2} \quad \text{(Q-ARROW)}$$

$$\frac{S_1 \equiv T_1 \qquad S_2 \equiv T_2}{\forall X{<:}S_1.S_2 \equiv \forall X{<:}T_1.T_2} \quad \text{(Q-ALL)}$$

$$\frac{S_2 \equiv T_2}{\lambda X{::}K_1.S_2 \equiv \lambda X{::}K_1.T_2} \quad \text{(Q-ABS)}$$

$$\frac{S_1 \equiv T_1 \qquad S_2 \equiv T_2}{S_1\ S_2 \equiv T_1\ T_2} \quad \text{(Q-APP)}$$

$$(\lambda X{::}K_{11}.T_{12})\ T_2 \equiv [X \mapsto T_2]T_{12} \quad \text{(Q-APPABS)}$$

続く...

▲図 31-1.　高階の有界量化（$F^{\omega}_{<:}$）

部分型付け $\boxed{\Gamma \vdash S <: T}$

$$\frac{\Gamma \vdash S <: U \qquad \Gamma \vdash U <: T \qquad \Gamma \vdash U :: K}{\Gamma \vdash S <: T} \quad \text{(S-TRANS)}$$

$$\frac{\Gamma \vdash S :: *}{\Gamma \vdash S <: \mathrm{Top}} \quad \text{(S-TOP)}$$

$$\frac{\Gamma \vdash T_1 <: S_1 \qquad \Gamma \vdash S_2 <: T_2}{\Gamma \vdash S_1 \to S_2 <: T_1 \to T_2} \quad \text{(S-ARROW)}$$

$$\frac{X <: T \in \Gamma}{\Gamma \vdash X <: T} \quad \text{(S-TVAR)}$$

$$\frac{\Gamma \vdash U_1 :: K_1 \qquad \Gamma, X <: U_1 \vdash S_2 <: T_2}{\Gamma \vdash \forall X <: U_1 . S_2 <: \forall X <: U_1 . T_2} \quad \text{(S-ALL)}$$

$$\frac{\Gamma, X <: \mathrm{Top}[K_1] \vdash S_2 <: T_2}{\Gamma \vdash \lambda X :: K_1 . S_2 <: \lambda X :: K_1 . T_2} \quad \text{(S-ABS)}$$

$$\frac{\Gamma \vdash S_1 <: T_1}{\Gamma \vdash S_1\ U <: T_1\ U} \quad \text{(S-APP)}$$

$$\frac{\Gamma \vdash S :: K \qquad \Gamma \vdash T :: K \qquad S \equiv T}{\Gamma \vdash S <: T} \quad \text{(S-EQ)}$$

型付け $\boxed{\Gamma \vdash t : T}$

$$\frac{x : T \in \Gamma}{\Gamma \vdash x : T} \quad \text{(T-VAR)}$$

$$\frac{\Gamma \vdash T_1 :: * \qquad \Gamma, x : T_1 \vdash t_2 : T_2}{\Gamma \vdash \lambda x : T_1 . t_2 : T_1 \to T_2} \quad \text{(T-ABS)}$$

$$\frac{\Gamma \vdash t_1 : T_{11} \to T_{12} \qquad \Gamma \vdash t_2 : T_{11}}{\Gamma \vdash t_1\ t_2 : T_{12}} \quad \text{(T-APP)}$$

$$\frac{\Gamma, X <: T \vdash t_2 : T_2}{\Gamma \vdash \lambda X <: T . t_2 : \forall X <: T . T_2} \quad \text{(T-TABS)}$$

$$\frac{\begin{array}{c}\Gamma \vdash t_1 : \forall X <: T_{11} . T_{12} \\ \Gamma \vdash T_2 <: T_{11}\end{array}}{\Gamma \vdash t_1\ [T_2] : [X \mapsto T_2] T_{12}} \quad \text{(T-TAPP)}$$

$$\frac{\Gamma \vdash t : S \qquad \Gamma \vdash S <: T \qquad \Gamma \vdash T :: *}{\Gamma \vdash t : T} \quad \text{(T-SUB)}$$

▲ 図 31-1.　高階の有界量化（$F^{\omega}_{<:}$）、続き

演習 31.2.1. [★]　$\mathrm{Id} = \lambda X.X$ および

$$\Gamma = B <: \mathrm{Top},\ A <: B,\ F <: \mathrm{Id}$$

と定義すると、以下の部分型付け判断式のいずれが導出可能だろうか。

$$\begin{array}{lllll}
\Gamma & \vdash & A & <: & \mathrm{Id}\ B \\
\Gamma & \vdash & \mathrm{Id}\ A & <: & B \\
\Gamma & \vdash & \lambda X.X & <: & \lambda X.\mathrm{Top} \\
\Gamma & \vdash & \lambda X.\ \forall Y <: X.\ Y & <: & \lambda X.\ \forall Y <: \mathrm{Top}.\ Y \\
\Gamma & \vdash & \lambda X.\ \forall Y <: X.\ Y & <: & \lambda X.\ \forall Y <: X.\ X \\
\Gamma & \vdash & F\ B & <: & B \\
\Gamma & \vdash & B & <: & F\ B \\
\Gamma & \vdash & F\ A & <: & F\ B \\
\Gamma & \vdash & \forall G <: (\lambda Y.\mathrm{Top} \to Y).\ G\ A & <: & \forall G <: (\lambda Y.\mathrm{Top} \to Y).\ \mathrm{Top} \to B \\
\Gamma & \vdash & \forall G <: (\lambda Y.\mathrm{Top} \to Y).\ G\ A & <: & \forall G <: (\lambda Y.\mathrm{Top} \to Y).\ G\ B \\
\Gamma & \vdash & \mathrm{Top}[* \Rightarrow *] & <: & \mathrm{Top}[* \Rightarrow * \Rightarrow *]
\end{array}$$

□

31.3 性質

$F^{\omega}_{<:}$（の近縁種）の基本的な性質（簡約時の型の保存、進行、最小型付けを含む）の証明は、この章の最初に参照した論文で見つけることができる。これらの証明はもちろん、部分型付け、有界量化、型演算子によって引き起こされる問題のすべてを、個別に対処しなくてはならない。それらに加え、構文主導の別の部分型付け規則を定義しようとすると、さらに話が複雑になる。部分型付け導出において、(28.3 節で見たような）型変数規則の場合だけでなく、型等価性の規則（S-Eq 規則）も、本質的に推移律と組み合わせて用いることができるのである。

例えば、Γ = X<:Top, F<:λY.Y という文脈では、Γ ⊢ F X <: X という部分型付け判断式は、以下のようにして証明できる（カインドについては省略）。

$$\dfrac{\dfrac{\dfrac{}{\Gamma \vdash \mathsf{F} <: \lambda\mathsf{Y}.\mathsf{Y}}\ \textsc{S-TVar}}{\Gamma \vdash \mathsf{F\,X} <: (\lambda\mathsf{Y}.\mathsf{Y})\,\mathsf{X}}\ \textsc{S-App} \qquad \dfrac{}{\Gamma \vdash (\lambda\mathsf{Y}.\mathsf{Y})\,\mathsf{X} <: \mathsf{X}}\ \textsc{S-Eq}}{\Gamma \vdash \mathsf{F\,X} <: \mathsf{X}}\ \textsc{S-Trans}$$

さらに、この相互作用は型式を正規形に簡約するだけでは避けられないことに注意されたい。なぜなら、F A という式は簡約基ではないためである。部分型検査時に変数 F を上界 λY.Y に**昇格**して、はじめてこれは簡約基になるのである。解決法は、部分型検査のはじめに型式を一度正規化し、昇格操作時に必要に応じて再度正規化するというものである。

31.4 注記

$F^{\omega}_{<:}$ の背後にあるアイディアの多くは Cardelli、特に論文 "*Structural Subtyping and the Notion of Power Type*" [76] による。部分型関係の型演算子への拡張は Cardelli [78] と Mitchell [320] で提案された。初期の意味論的モデルは Cardelli and Longo [85] によって部分同値関係を用いて与えられた。Compagnoni and Pierce [108] は、$F^{\omega}_{<:}$ を交差型で拡張した体系のモデルを与えた。再帰型を含むより強力なモデルは Bruce and Mitchell [65] によって与えられた。これに関連するモデルは Abadi and Cardelli [4] にも見られる。

本章で定義した $F^{\omega}_{<:}$ の一種に関する、基本的なメタ理論的性質は、Pierce and Steffen [371] および、独立して（主要な議論の一つを単純化する巧妙な証明技法を用いて）Compagnoni [107] によって与えられた。Compagnoni の技法は、Abadi and Cardelli [4] で、核となる項言語にラムダ計算ではなくオブジェクト計算を用いた $F^{\omega}_{<:}$ の変種に対しても用いられた。

本章では型演算子間の部分型付けを各引数において定義したが、これは異なる型演算子を異なる引数に適用した場合（F S <: G T）に一般化することもできる。ただし、そのためにはカインドシステムを詳細化し型演算子の**極性**を追跡するようにする必要がある。演算子 F は、S <: T のとき常に F S <: F T であるならば、**共変**であるという。同様に、S <: T であるとき常に F T <: F S であるならば、**反変**であるという。これらの性質を反映した以下の新たな部分型付け規則を導入するとする。

$$\dfrac{\Gamma \vdash \mathsf{S} <: \mathsf{T} \qquad \mathsf{F}\ \text{は共変}}{\Gamma \vdash \mathsf{F\,S} <: \mathsf{F\,T}}$$

$$\dfrac{\Gamma \vdash \mathsf{S} <: \mathsf{T} \qquad \mathsf{F}\ \text{は反変}}{\Gamma \vdash \mathsf{F\,T} <: \mathsf{F\,S}}$$

すると、F <: G かつ S <: T で G が共変ならば、(推移律により）F S <: G T である。これを完全に

機能するようにするためには、型変数にもその極性の印を付け、高階量化子が特定の極性の演算子の上だけを動くように制限する必要がある。$F^{\omega}_{<:}$ に極性を導入したバージョンは、Cardelli [78] や Steffen [444]、Duggan and Compagnoni [141] によって検討されている。

ここで示した $F^{\omega}_{<:}$ の形式化の別の一般化として、有界でない型演算子 $\lambda X :: K_1 . T_2$ を、**有界型演算子**へと一般化することが挙げられる。これは魅力的な一般化である。というのは、System F に部分型付けを加えて $F_{<:}$ を形成した際に、量化子を有界量化子へと一般化した方法と一致しているためである。その一方、これは体系を大幅に複雑化してしまう。なぜなら、$\forall X <: T_1 . K_2$ といったカインドを含むようカインドシステムも一般化しなくてはならないが、これは回りまわって、カインド規則と部分型付け規則の間の相互依存性を発生させてしまう。この相互依存を解消するには相当の議論が必要になる。Compagnoni and Goguen [105; 106] を参照されたい。

$F^{\omega}_{<:}$ の依存型による拡張は、Chen and Longo [97] および Zwanenburg [507] によって研究されている。

事例：純粋関数的オブジェクト

本章で扱う例はレコード（図 11-7)、数（図 8-2)、多相更新（図 32-1）を持つ $F^{\omega}_{<:}$（図 31-1）の項である。対応する OCaml 実装は fullupdate である。

本書の最後の事例として、**存在型オブジェクトモデル**の説明を続ける。このモデルは 291 ページで簡単に紹介した。291 ページでは、存在型パッケージを単純なオブジェクトとみなす方法を示し、そのスタイルの抽象化の性質を、存在型を用いてより伝統的な抽象データ型を実装する方法と比較した。この章では、以前の数章で説明した道具（型演算子、高階部分型付け）と 32.7 節で導入する新しい**多相更新**の機能を用いて、これまでの単純な存在型オブジェクトを、クラスや継承を含む一人前のオブジェクト指向プログラミングの柔軟性をもたらすイディオムへと拡張する。

32.1 単純オブジェクト

まずは、291 ページの純粋関数的な Counter オブジェクトの型を思い出そう。

```
Counter = {∃X, {state:X, methods:{get:X→Nat,inc:X→X}}};
```

この型の要素は、隠蔽された**状態型** X、型 X の**状態**、型 {get:X→Nat,inc:X→X} の**メソッド**のレコードからなる存在型パッケージである。

この章の最初の数節では、型 {x:Nat} をすべてのオブジェクトの表現型として用いる。（32.8 節で、複数のインスタンス変数を持つオブジェクトや、インスタンス変数を追加するクラスを定義する方法を示す。）この内部状態の型について述べるときは、一貫して略記 CounterR を用いる。

```
CounterR = {x:Nat};
```

カウンタオブジェクトは型 Counter の要素である。これは存在型の導入規則（図 24-1 の T-Pack）に従って定義される。

```
c = {*CounterR,
      {state = {x=5},
       methods = {get = λr:CounterR. r.x,
                  inc = λr:CounterR. {x=succ(r.x)}}}} as Counter;
▸ c : Counter
```

Counter のメソッドを呼び出すには、まずアンパッケージ化し、次にメソッド群の中から適切なフィールドを選び、それを状態に適用する。

```
sendget = λc:Counter.
            let {X,body} = c in
            body.methods.get(body.state);
▸ sendget : Counter → Nat
```

そして（単なる数ではなく新しいオブジェクトを返す inc の場合には）最後に、元のオブジェクトと同じ表現型とメソッド群で、返り値を再び新しいオブジェクトにパッケージ化する。

```
sendinc = λc:Counter.
            let {X,body} = c in
              {*X,
               {state = body.methods.inc(body.state),
                methods = body.methods}} as Counter;
▸ sendinc : Counter → Counter
```

これらの基本関数は、Counter オブジェクトを操作するもっと複雑な項を構築するために使うことができる。

```
addthree = λc:Counter. sendinc (sendinc (sendinc c));
▸ addthree : Counter → Counter
```

32.2 部分型付け

この存在型によるオブジェクトの表現は、オブジェクト型の間に期待される部分型の包含関係が、存在型とレコードの部分型付け規則から直ちに導けるという好ましい特徴を持つ。これを確かめるには、（図 26-3 から）存在型の部分型付け規則を思い出すとよい[†1]。

$$\frac{\Gamma, X{<:}U \vdash S_2 <: T_2}{\Gamma \vdash \{\exists X{<:}U, S_2\} <: \{\exists X{<:}U, T_2\}} \qquad \text{(S-SOME)}$$

この規則より直ちに次のことがわかる。もし、Counter より多くのメソッドを持つオブジェクト型、例えば

```
ResetCounter =
  {∃X, {state:X, methods:{get: X→Nat, inc:X→X, reset:X→X}}};
```

を定義したならば、その型は Counter の部分型、つまり ResetCounter <: Counter である。これにより、リセットカウンタオブジェクト

```
rc = {*CounterR,
       {state = {x=0},
        methods = {get = λr:CounterR. r.x,
                   inc = λr:CounterR. {x=succ(r.x)},
                   reset = λr:CounterR. {x=0}}}} as ResetCounter;
▸ rc : ResetCounter
```

を定義すると、包摂の規則を使って、例えば sendget、sendinc、addthree など Counter 上で定義された関数にこのオブジェクトを渡せる。

```
rc3 = addthree rc;
▸ rc3 : Counter
```

ただし、これを行うと型の情報が失われることに注意されたい。つまり、ここで rc3 の型はただの Counter であり、ResetCounter ではない。

[†1] ここではカーネル版の規則のみを用いる。フル版の能力は必要でない。実際、この章では**有界**存在型は全く必要ない。つまり、すべての存在型の上界は Top である。

32.3 有界量化

もちろん、包摂によるこの種の情報の損失が、まさに第 26 章で有界量化を導入した動機だった。しかしながら、有界量化それ自体ではまだ十分ではない。今抱えている問題を解決する手助けとするには、何らかの追加の機構で拡張する必要がある。

この理由は、有界量化を用いて sendinc の型を自明に詳細化すると ∀C<:Counter. C→C となることを考えればわかる。この型の sendinc があれば、addthree は

```
addthree = λC<:Counter. λc:C.
             sendinc [C] (sendinc [C] (sendinc [C] c));
▸ addthree : ∀C<:Counter. C → C
```

と書け、これを rc に適用すれば ResetCounter 型の結果が得られる。

```
rc3 = addthree [ResetCounter] rc;
▸ rc3 : ResetCounter
```

残念ながら、このような関数を書く方法は存在しない。というより、望み通りに振る舞い、かつこの型を持つ関数を書く方法は存在しない。もちろん、この型に属する**恒等**関数を書くことはできる。

```
wrongsendinc = λC<:Counter. λc:C. c;
▸ wrongsendinc : ∀C<:Counter. C → C
```

しかしながら、上記の sendinc の本当の実装に対し、有界な型抽象を先頭に加えて詳細化を試みても、その項は型検査に失敗してしまう。

```
sendinc =
   λC<:Counter. λc:C.
     let {X,body} = c in
       {*X,
        {state = body.methods.inc(body.state),
         methods = body.methods}}
       as C;
▸ Error: existential type expected
```

問題は最後の行にある。注釈 as C は型検査器に「ここで作ろうとしている存在型パッケージに存在型 C を用いよ」と述べている。しかしながら C は存在型**ではなく**、型変数であるため、これは不正である。これは、今までに定義した型付け規則につまらない制限があるせいではない。例えば、ある存在型のすべての部分型は必ず存在型であることを型付け規則が「知らない」ためではない (Counter の部分型 C は必ず存在型である)。それどころか、存在型パッケージ

```
{*X,
 {state = body.methods.inc(body.state),
  methods = body.methods}}
```

に型 C を与えるのは実際**誤り**でありうる。例えば、型

```
{∃X, {state:X, methods:{get:X→Nat,inc:X→X}, junk:Bool}}
```

は Counter の部分型であることに注目せよ。しかし上の存在型パッケージは確実にこの型を持たない。フィールド junk を持たないためである。つまり上記の sendinc の本体は、仮に型付け規則にそれがわかるとしても、Counter のすべての部分型 C について型 C を「本当に」持つわけではない。事実、純粋な $F_{<:}$ では、∀C<:T.C→C の形を持つ型は 恒等関数**のみ**を含むことを示せる（例えば $F_{<:}$ の表示的意味論を用いる。Robinson and Tennent [422] を参照）。

この $F_{<:}$ の欠点を解決する方法は幾つか提案されている。一つの方法は、$F_{<:}$ から $F^{\omega}_{<:}$ に移り、高階の有界量化を用いてもっと詳細化された型を sendinc のような関数に与えることである。他の方法は、型 ∀C<:Counter.C→C は変えずに、この型に属する意味のある項を構築できるよう言語に機能を追加することである。最後に、単に言語に参照セルを加える方法もあるが、その方法は既に第 27 章で採用してしまった。本章の目的は、純粋関数的な世界でできること（また、それを行う方法）を試すことである。

以降の説明では、上述の二つの方法を組み合わせる。$F^{\omega}_{<:}$ を用いて前節で述べたオブジェクト型の量化に関する問題を取り扱い、それに関連してインスタンス変数を扱う際に生じる問題を、**多相レコード更新**という新しいプリミティブ（32.7 節で定義する）で扱う（32.8 節）。

32.4 インターフェイス型

型演算子を使えば、Counter を次の二つの組み合わせで表現できる。

```
Counter = Object CounterM;
```

ここで

```
CounterM = λR. {get: R→Nat, inc:R→R};
```

はカウンタオブジェクトに特化したメソッドインターフェイスを表すカインド *⇒* の型演算子であり、

```
Object = λM::*⇒*. {∃X, {state:X, methods:M X}};
```

はすべてのオブジェクト型に共通する構造を表すカインド (*⇒*)⇒* の型演算子である。このように再構成することで、オブジェクトごとに変化する部分（メソッドインターフェイス。ここでは部分型付けを許したい）と、変化しない共通の部分（存在型によるパッケージ化と、状態とメソッドの二つ組。再パッケージ化の邪魔になるのでここでは部分型付けを禁止したい）を分離できる。

このように分離するには型演算子の有界量化が必要である。それを用いれば、オブジェクト型からメソッドインターフェイスを抜き出せるためである。インターフェイスは存在量化された状態型 X に言及しているが、メソッドインターフェイス自体を X に関して抽象化すればよい。それゆえ、このインターフェイスは「引数を取る引数」になる。このように引数が反復的になる特徴は、Object のカインドに反映されている。また、Object CounterM の型適用を簡約するステップにも反映されている。最初に、CounterM が Object の本体の中に代入され、

```
{∃X, {state:X, methods:(λR. {get: R→Nat, inc:R→R}) X}}
```

が得られる。次に X が CounterM の本体の中に代入され、

```
{∃X, {state:X, methods:{get:X→Nat,inc:X→X}}}
```

が得られる。

ResetCounter を同じように分割すると、

```
ResetCounterM = λR. {get: R→Nat, inc:R→R, reset:R→R};
ResetCounter = Object ResetCounterM;
```

以前のように

```
ResetCounter <: Counter
```

となるだけでなく、型演算子の間の部分型付けの規則により、

ResetCounterM <: CounterM

となる。つまり、オブジェクト型を一般的なひな形と特定のインターフェイスに分割すれば、完全なオブジェクト型の間の部分型関係とは別物の、意味のある「インターフェイス部分型付け」の概念が得られるのである。

インターフェイス部分型付けは、**マッチング**のアイディアと、概念的も技術的にも密接に関連している。マッチングは Bruce et al. [66] によって導入され、Abadi and Cardelli [3; 4] によってより深く研究された。

32.5 オブジェクトにメッセージを送る

ここで、32.3 節の誤った sendinc は、Counter の部分型ではなく CounterM の部分**インターフェイス**で型抽象することで修復できる。

```
sendinc =
  λM<:CounterM. λc:Object M.
    let {X, b} = c in
      {*X,
       {state = b.methods.inc(b.state),
        methods = b.methods}}
       as Object M;
▸ sendinc : ∀M<:CounterM. Object M → Object M
```

直観的には、sendinc の型は次のように読める。「カウンタのインターフェイスを詳細化したインターフェイスを与え、次にそのインターフェイスを持つオブジェクトを与えると、同じインターフェイスを持つ別のオブジェクトを返す。」

演習 32.5.1. [★] なぜこの sendinc は正しく型付けされ、以前の sendinc はそうでないのか。 □

カウンタオブジェクトやリセットカウンタオブジェクトのメソッドを呼び出すには、多相的なメソッド呼び出し関数を適切なインターフェイスシグネチャ CounterM や ResetCounterM で具体化する（sendget と sendreset は同様に定義されていると仮定する）。

```
sendget [CounterM] (sendinc [CounterM] c);
▸ 6 : Nat

sendget [ResetCounterM]
        (sendreset [ResetCounterM]
           (sendinc [ResetCounterM] rc));
▸ 0 : Nat
```

演習 32.5.2. [推奨, ★★] sendget と sendreset を定義せよ。 □

32.6　単純クラス

ここからはクラスについて考える。まずは（第 18 章と同じように）`self` を持たない**単純なクラス**から始めよう。

18.6 節では、単純なクラスを状態からオブジェクトへの関数で定義した（命令的なオブジェクト表現のためのもので、オブジェクトはメソッドのレコードだった）。これは、同じメソッドを持つものの、インスタンス変数の集合は新たに確保して別々に持つ、複数のオブジェクトをこしらえる方法だった。この章では、オブジェクトは単なるメソッドのレコードではない。つまり、オブジェクトは表現型と状態も含む。他方では、これは純粋関数的なモデルであるため、それぞれのメソッドは状態を引数に取る（そして、必要に応じて更新された状態を持つオブジェクトを返す）。このためオブジェクトを生成する時点では状態をクラスに渡す必要はない。実際、ここでのクラスは、すべてのオブジェクトが同じ表現型を持つという当初からの仮定の下では、単にメソッドのレコードであるとみなせる。

```
counterClass =
   {get = λr:CounterR. r.x,
    inc = λr:CounterR. {x=succ(r.x)}}
 as {get: CounterR→Nat, inc:CounterR→CounterR};
▸ counterClass : {get:CounterR→Nat, inc:CounterR→CounterR}
```

または、`CounterM` 型演算子を用いれば注釈をより短く書ける。

```
counterClass =
   {get = λr:CounterR. r.x,
    inc = λr:CounterR. {x=succ(r.x)}}
   as CounterM CounterR;
▸ counterClass : CounterM CounterR
```

このようなクラスのインスタンスを構築するには、状態の初期値を与え、その状態とメソッド群（つまり、クラス）をオブジェクトにパッケージ化する。

```
c = {*CounterR,
      {state = {x=0},
       methods = counterClass}}
     as Counter;
▸ c : Counter
```

サブクラスを定義するには単に新しいメソッドのレコードを構築すればよい。ここで、幾つかのフィールドは前に定義したものからコピーすることになる。

```
resetCounterClass =
   let super = counterClass in
   {get = super.get,
    inc = super.inc,
    reset = λr:CounterR. {x=0}}
   as ResetCounterM CounterR;
▸ resetCounterClass : ResetCounterM CounterR
```

これらの単純なクラスを一般化し、第 18 章の最後に扱ったのと同様の例を扱えるようにするためには、追加で二つのものが必要である。サブクラスにおいて新しいインスタンス変数を追加する能力と、`self` の扱いである。次の二つの節では前者を取り扱う。そして 32.9 節 で `self` を扱い、本章を終える。

32.7 多相更新

インスタンス変数をクラスに追加するには新しい機構が必要である。レコードのフィールドを置き換える**多相更新**のためのプリミティブと、それに対応するレコード型の詳細化である。これらの機能は、次のような事実から必要性が生じる。クラス間におけるインスタンス変数の違いを認めると、サブクラスのインスタンス変数に関してスーパークラスが多相的になるのである。どのようにしてこれが起こるか見てみよう。

resetCounterClass のサブクラスを次のように定義したいと仮定する。カウンタの現在の値を保存する backup メソッドを加え、reset の振る舞いを 定数の初期値ではなく backup で保存した値に戻すように変更する。保存した値を保持するには、状態型を {x:Nat} から {x:Nat, old:Nat} に拡張する必要がある。表現におけるこの違いは、直ちに技術的困難をもたらす。backupCounterClass を定義するときに、inc メソッドを resetCounterClass から再利用できることは、このメソッドが両方のクラスで同じように振る舞うことに依存している。しかしながら、インスタンス変数の集合が異なる場合、それは全く同じに振る舞うわけ**ではない**。ResetCounter の inc メソッドは 型 {x:Nat} の状態を期待し、同じ型の新しい状態を返すが、一方 BackupCounter の inc は 型 {x:Nat,old:Nat} の状態を期待し、返す。

この問題を解決するには、次のことに気づけば十分である。inc メソッドは必ずしも状態型が {x:Nat} や {x:Nat,old:Nat} **である**ことを知っている必要はなく、ただ状態がインスタンス変数 x を**含む**ことを知っていればよい。言い換えれば、これら二つのメソッドは、両方に型 ∀S<:{x:Nat}.S→S を与えれば統一できる。

ここで状態に関して、32.3 節でオブジェクト全体がそうであったのと同じ困難が生じる。現在の言語では、型 ∀S<:{x:Nat}.S→S は恒等関数しか要素に持たないのである。今回も、この問題を扱うにはより精密な有界量化の形式を許容する何らかの仕組みが必要である。ここで最も直接的な仕組みは、レコードのフィールドの**多相更新**のためのプリミティブを追加することである[†2]。もし r が型 T のフィールド x を持つレコードで、t が型 T の項ならば、r←x=t と書いて「フィールド x が値 t を持つことを除き、r とちょうど同じレコード」を意味するものとする。これは純粋関数的な更新操作であることに注意されたい。つまり、これは r を更新するのではなく、異なる x フィールドを持つ複製を作る。

このレコード更新プリミティブを用いれば、inc のメソッド本体で意図したような振る舞いを表現する関数は概ね次の通り書ける。

```
f = λX<:{a:Nat}. λr:X. r←a = succ(r.a);
```

しかし、少し注意しなければならない。多相更新演算子の安易な型付け規則はおそらく次のようなものになる。

$$\frac{\Gamma \vdash \mathtt{r} : \mathtt{R} \qquad \Gamma \vdash \mathtt{R} <: \{\mathtt{l}_j\mathtt{:T}_j\} \qquad \Gamma \vdash \mathtt{t} : \mathtt{T}_j}{\Gamma \vdash \mathtt{r}\leftarrow\mathtt{l}_j\mathtt{=t} : \mathtt{R}}$$

しかしこの規則は健全ではない。例えば、仮に

```
s = {x={a=5,b=6},y=true};
```

とすると、s : {x:{a:Nat,b:Nat},y:Bool} であり、かつ {x:{a:Nat,b:Nat},y:Bool} <: {x:{a:Nat}} であるため、上の規則から次の導出

†2 これまでと同じく、同様の効果を達成する方法は実際には幾つかある。別のプリミティブを導入するか（幾つかは 32.10 節に記載している）、Pierce and Turner [373] のように多相性を使うことである。後者はより記法が重いものの、理論的にはより初等的な選択肢である。ここで用いる方法は、その簡単さと、以降で用いる例に自然に適合することから選んだ。

→ ∀ <: Top {} ←　　　$F_{<:}$（図 26-1）とレコード（図 11-7）に基づく

新しい構文形式

t	::=	...	項：
		$\{\iota_i\,\mathtt{l}_i\mathtt{=t}_i\ ^{i\in 1..n}\}$	レコード
		$\mathtt{t\leftarrow l=t}$	フィールド更新
T	::=	...	型：
		$\{\iota_i\,\mathtt{l}_i\mathtt{:T}_i\ ^{i\in 1..n}\}$	レコードの型
ι	::=	#	非変な（更新可能）フィールド
		omitted	共変な（更新不能）フィールド

新しい評価規則　$\boxed{\mathtt{t} \longrightarrow \mathtt{t}'}$

$$\frac{\mathtt{t}_1 \longrightarrow \mathtt{t}_1'}{\mathtt{t}_1\mathtt{\leftarrow l}_i\mathtt{=t}_2 \longrightarrow \mathtt{t'}_1\mathtt{\leftarrow l}_i\mathtt{=t}_2} \quad \text{(E-UPDATE1)}$$

$$\frac{\mathtt{t}_2 \longrightarrow \mathtt{t}_2'}{\mathtt{v}_1\mathtt{\leftarrow l}_i\mathtt{=t}_2 \longrightarrow \mathtt{v}_1\mathtt{\leftarrow l}_i\mathtt{=t'}_2} \quad \text{(E-UPDATE2)}$$

$$\{\iota_j\mathtt{l}_j\mathtt{=v}_j\ ^{j\in 1..n}\}\mathtt{\leftarrow l}_i\mathtt{=v} \longrightarrow \{\iota_j\mathtt{l}_j\mathtt{=v}_j\ ^{j\in 1..i-1}, \iota_i\mathtt{l}_i\mathtt{=v}, \iota_k\mathtt{l}_k\mathtt{=v}_k\ ^{k\in i+1..n}\} \quad \text{(E-UPDATEV)}$$

$$\{\iota_i\,\mathtt{l}_i\mathtt{=v}_i\ ^{i\in 1..n}\}\mathtt{.l}_j \longrightarrow \mathtt{v}_j \quad \text{(E-PROJRCD)}$$

$$\frac{\mathtt{t}_j \longrightarrow \mathtt{t}_j'}{\{\iota_i\,\mathtt{l}_i\mathtt{=v}_i\ ^{i\in 1..j-1}, \iota_j\,\mathtt{l}_j\mathtt{=t}_j, \iota_k\,\mathtt{l}_k\mathtt{=t}_k\ ^{k\in j+1..n}\} \longrightarrow \{\iota_i\,\mathtt{l}_i\mathtt{=v}_i\ ^{i\in 1..j-1}, \iota_j\,\mathtt{l}_j\mathtt{=t}_j', \iota_k\,\mathtt{l}_k\mathtt{=t}_k\ ^{k\in j+1..n}\}} \quad \text{(E-RCD)}$$

新しい部分型付け規則　$\boxed{\Gamma \vdash \mathtt{S} <: \mathtt{T}}$

$$\Gamma \vdash \{\iota_i\,\mathtt{l}_i\mathtt{:T}_i\ ^{i\in 1..n+k}\} <: \{\iota_i\,\mathtt{l}_i\mathtt{:T}_i\ ^{i\in 1..n}\} \quad \text{(S-RCDWIDTH)}$$

$$\frac{\text{各 } i \text{ に対して } \Gamma \vdash \mathtt{S}_i <: \mathtt{T}_i \qquad \iota_i = \#\ \text{ならば}\ \Gamma \vdash \mathtt{T}_i <: \mathtt{S}_i}{\Gamma \vdash \{\iota_i\,\mathtt{l}_i\mathtt{:S}_i\ ^{i\in 1..n}\} <: \{\iota_i\,\mathtt{l}_i\mathtt{:T}_i\ ^{i\in 1..n}\}} \quad \text{(S-RCDDEPTH)}$$

$$\Gamma \vdash \{\ldots\#\mathtt{l}_i\mathtt{:S}_i\ldots\} <: \{\ldots\mathtt{l}_i\mathtt{:S}_i\ldots\} \quad \text{(S-RCDVARIANCE)}$$

新しい型付け規則　$\boxed{\Gamma \vdash \mathtt{t} : \mathtt{T}}$

$$\frac{\text{各 } i \text{ に対して } \Gamma \vdash \mathtt{t}_i : \mathtt{T}_i}{\Gamma \vdash \{\iota_i\,\mathtt{l}_i\mathtt{=t}_i\ ^{i\in 1..n}\} : \{\iota_i\,\mathtt{l}_i\mathtt{:T}_i\ ^{i\in 1..n}\}} \quad \text{(T-RCD)}$$

$$\frac{\Gamma \vdash \mathtt{t}_1 : \{\iota_i\,\mathtt{l}_i\mathtt{:T}_i\ ^{i\in 1..n}\}}{\Gamma \vdash \mathtt{t}_1\mathtt{.l}_j : \mathtt{T}_j} \quad \text{(T-PROJ)}$$

$$\frac{\Gamma \vdash \mathtt{r} : \mathtt{R} \qquad \Gamma \vdash \mathtt{R} <: \{\#\mathtt{l}_j\mathtt{:T}_j\} \qquad \Gamma \vdash \mathtt{t} : \mathtt{T}_j}{\Gamma \vdash \mathtt{r\leftarrow l}_j\mathtt{=t} : \mathtt{R}} \quad \text{(T-UPDATE)}$$

▲図 32-1.　多相更新

$$\mathtt{s\leftarrow x=\{a=8\} : \{x:\{a:Nat,b:Nat\},y:Bool\}}$$

が許容されるが、s←x={a=8} は {x={a=8},y=true} に簡約されるため、この導出は誤りだろう。

この問題の原因は、フィールド x において深さ部分型付けを使用して {x:{a:Nat,b:Nat}, y:Bool} <: {x:{a:Nat}} を導出したことである。更新されうるフィールドにおいては深さ部分型付けを認めるべきではない。これは、そのようなフィールドを特別な印 # で注釈するという簡単な間に合わせの処置で達成できる。

これら「更新可能レコード」と更新操作自体の規則を図 32-1 で与える。レコード型の構文を詳細化し、すべてのフィールドが、深さ部分型付けを許すかどうかを示す**変性**タグで注釈されるようにする。# はこのフィールドにおいて部分型付けを禁止し、空文字列は部分型付けを許す（ここで空文字列を選んだため、印の付いていないレコードはこれまでと同様に振る舞うことになる）。深さ部分型付けの規則 S-RcdDepth は、印の付いていないフィールドにおいてのみ部分型付けを許すように改良する。最後に、フィールドに付けた印を # から空文字列に変えることを許す部分型付け規則 S-RcdVariance を追加する。これは言い換えれば、そのフィールドが更新可能であることを「忘れる」ことを許す規則である。この更新プリミティブの型付け規則は、置き換えられるフィールド

に # の印が付いていることを要求する。規則 E-UPDATEV が更新操作を実装する。

上記の関数 f は次の通り書けるようになる。

```
  f = λX<:{#a:Nat}. λr:X. r←a = succ(r.a);
▸ f : ∀X<:{#a:Nat}. X → X
```

そして次のように使える。

```
  r = {#a=0, b=true};
  f [{#a:Nat,b:Bool}] r;
▸ {#a=1, b=true} : {#a:Nat, b:Bool}
```

更新操作の健全性は、詳細化された部分型関係に関する次の観察に基づく。

事実 32.7.1. $\vdash$ R <: {#l:T_1} ならば、R = {...#l:R_1...} であり、ここで $\vdash$ R_1 <: T_1 かつ $\vdash$ T_1 <: R_1. □

演習 32.7.2. [推奨, ★★★] この計算体系で、最小型付けの性質は成り立つか。そうならば、証明せよ。そうでないならば、それを修正する方法を示せ。 □

32.8 インスタンス変数の追加

前節の機能を使えば、内部状態の型に関して多相的な counterClass を書ける。

```
  CounterR = {#x:Nat};

  counterClass =
     λR<:CounterR.
        {get = λs:R. s.x,
         inc = λs:R. s←x=succ(s.x)}
        as CounterM R;
▸ counterClass : ∀R<:CounterR. CounterM R
```

新しい counterClass からオブジェクトを構築するには、単に CounterR を表現型として与えてやればよい。

```
  c = {*CounterR,
       {state = {#x=0},
        methods = counterClass [CounterR]}}
      as Object CounterM;
▸ c : Counter
```

この新しいクラスから構築したオブジェクトは以前のものと同じ型 Counter = Object CounterM を持つことに注意されたい。インスタンス変数の扱いの変更は専らクラス内部で起こっているためである。以前のメソッド呼び出し関数も、新しいクラスからインスタンス化したオブジェクトで使える。

resetCounterClass も同じ形式で書ける。

```
resetCounterClass =
   λR<:CounterR.
     let super = counterClass [R] in
       {get = super.get,
        inc = super.inc,
        reset = λs:R. s←x=0}
       as ResetCounterM R;
▸ resetCounterClass : ∀R<:CounterR. ResetCounterM R
```

最後に、今回は BackupCounterR の部分型で型抽象すれば、backupCounterClass を書ける。(それこそが、ここまでの全演習の要点だった)。

```
BackupCounterM = λR. {get:R→Nat,inc:R→R,reset:R→R,backup:R→R};
BackupCounterR = {#x:Nat,#old:Nat};
backupCounterClass =
   λR<:BackupCounterR.
   let super = resetCounterClass [R] in
    {get = super.get,
     inc = super.inc,
     reset = λs:R. s←x=s.old,
     backup = λs:R. s←old=s.x}
    as BackupCounterM R;
▸ backupCounterClass : ∀R<:BackupCounterR. BackupCounterM R
```

32.9 “self” を伴ったクラス

18.9 節では、クラスのメソッドがお互いを再帰的に参照できるようにする機構で命令的なクラスを拡張する方法について調べた。この拡張は純粋関数的な世界でも意味をなす。

まず counterClass を、同じ表現型 R に適合するメソッドの集まり self で抽象化する。

```
counterClass =
   λR<:CounterR.
   λself: Unit→CounterM R.
   λ_:Unit.
      {get = λs:R. s.x,
       inc = λs:R. s←x=succ(s.x)}
      as CounterM R;
```

18.11 節で見たように、クラスに与える Unit 引数は、オブジェクトのメソッドを生成する fix 操作において、評価を遅延するために用いられる。これに対応して、self の型は Unit 型の関数抽象を含む。

このクラスからオブジェクトを構築するには、関数 counterClass の不動点を取り unit に適用する。

```
c = {*CounterR,
      {state = {#x=0},
       methods = fix (counterClass [CounterR]) unit}}
     as Object CounterM;
▸ c : Counter
```

次に、set 操作を提供するサブクラスを、以下のインターフェイスを持つように定義する。

```
SetCounterM = λR. {get: R→Nat, set:R→Nat→R, inc:R→R};
```

setCounterClass の実装は set メソッドを定義し、self の set と get メソッドを inc メソッドの実装で用いる。

```
setCounterClass =
   λR<:CounterR.
   λself: Unit→SetCounterM R.
   λ_:Unit.
      let super = counterClass [R] self unit in
      {get = super.get,
       set = λs:R. λn:Nat. s←x=n,
       inc = λs:R. (self unit).set s (succ((self unit).get s))}
   as SetCounterM R;
```

最後に、本章のすべての機構を合わせれば、自身が呼び出された回数を数える set 操作を持つ計器付きカウンタクラスをサブクラスとして構築できる。

```
InstrCounterM =
  λR. {get: R→Nat, set:R→Nat→R, inc:R→R, accesses:R→Nat};

InstrCounterR = {#x:Nat,#count:Nat};

instrCounterClass =
   λR<:InstrCounterR.
   λself: Unit→InstrCounterM R.
   λ_:Unit.
      let super = setCounterClass [R] self unit in
      {get = super.get,
       set = λs:R. λn:Nat.
                let r = super.set s n in
                r←count=succ(r.count),
       inc = super.inc,
       accesses = λs:R. s.count}
   as InstrCounterM R;
```

inc の呼び出しを含めてアクセス回数が数え上げられていることに注意されたい。なぜなら inc が self の set メソッドを用いて実装されているためである。

まとめとして、計器付きカウンタオブジェクトを構築し、幾つかメッセージを送ってみよう。

```
ic = {*InstrCounterR,
       {state = {#x=0,#count=0},
        methods = fix (instrCounterClass [InstrCounterR]) unit}}
      as Object InstrCounterM;
▸ ic : Object InstrCounterM

sendaccesses [InstrCounterM] (sendinc [InstrCounterM] ic);
▸ 1 : Nat
```

演習 32.9.1. [推奨, ★★★] backup と reset のメソッドを追加する instrCounterClass のサブクラスを定義せよ。 □

32.10 注記

型付きラムダ計算による最初の「純粋関数的な」オブジェクトの解釈は、再帰的に定義されたレコードに基づいていた。これは Cardelli [73] により導入され、多くの変種が Kamin と Reddy [397; 249]、Cook と Palsberg [112]、Mitchell [320] によって研究された。型無しのものは、型無しのオブジェクト指向言語の表示的意味論でかなり効果的に用いられた。型付きのものは、個別のオブジェクト指向の例を表現するために用いられたが、型付きオブジェクト指向言語の統一的な解釈には困難が生じた。この方向性で最もうまくいったのは、Cook と彼の同僚の成果である（Cook, Hill, and Canning [114]; Canning, Cook, Hill, and Olthoff [70]; Canning, Cook, Hill, Olthoff, and Mitchell [71]）。

Pierce と Turner [373] が導入したオブジェクト表現は、存在型を持つが再帰型は持たない型システムにのみ依存していた。このことは Hofmann と Pierce [215] が、関数的な計算体系による、最初の、一様で型主導なオブジェクトの解釈を発表するきっかけとなった。同じ国際会議で、Bruce は関数的オブジェクト指向言語の意味論に関する論文 [59] を発表した。この意味論は最初、$F^{\omega}_{<:}$ の表示的モデルへの直接のマッピングの形で示されたが、最近、存在型と再帰型の両方に依存するオブジェクト表現に再構成された。一方、オブジェクトをラムダ計算により表現する難しさに苛立ちを感じ、Abadi と Cardelli はプリミティブなオブジェクトの計算体系を導入した [4]。しかしながら、後に Abadi, Cardelli, and Viswanathan [9] はこのオブジェクト計算の有界存在型と再帰型による忠実な表現を発見した。これらの発展は Bruce et al. [63] と Abadi and Cardelli [4] によってサーベイされている。

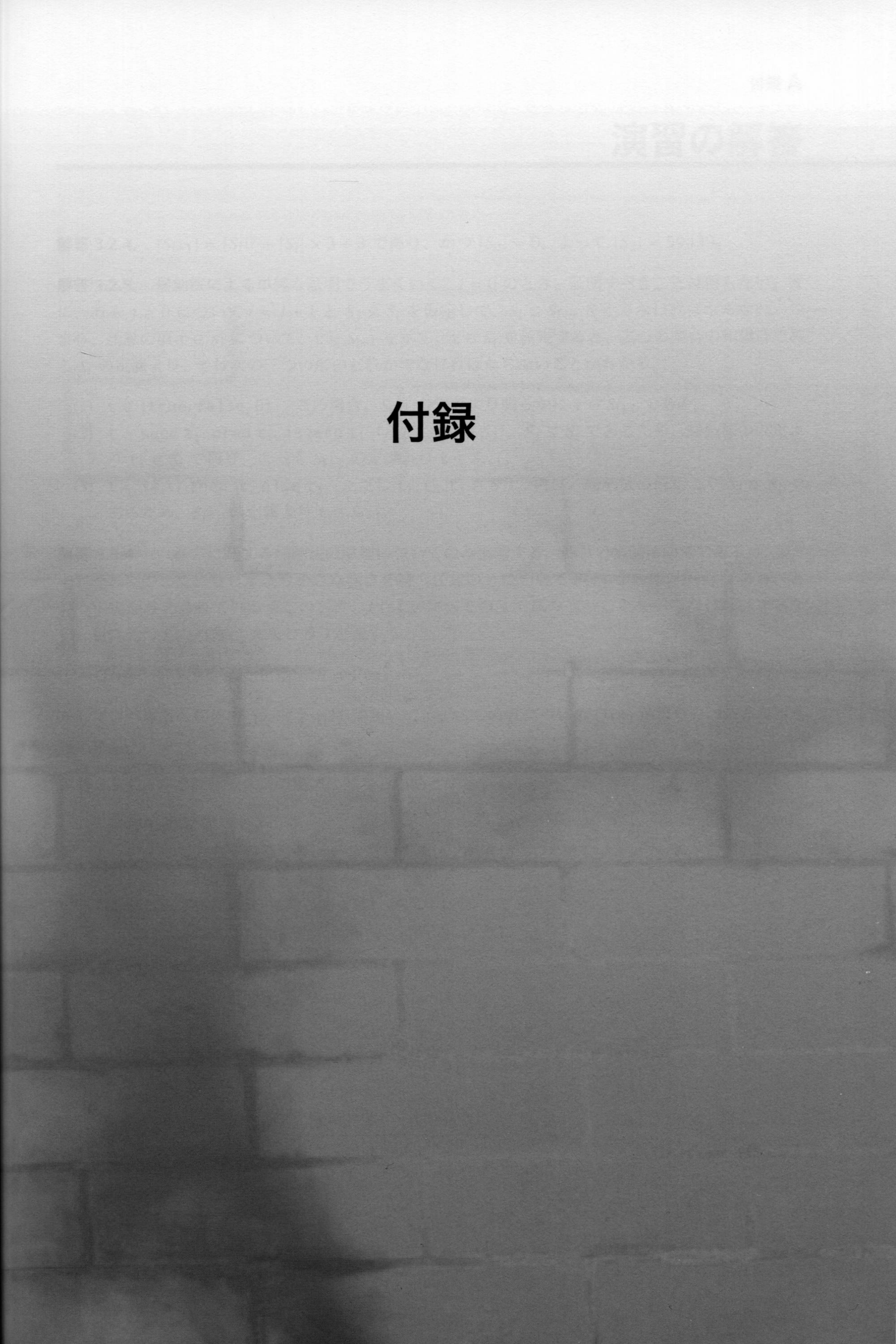
この章のオブジェクト表現は Compagnoni and Pierce [108] で**多重継承**、つまり一つ以上のスーパークラスを持つクラスを含むように拡張された。鍵となるアイディアは、**交差型**（15.7 節）による $F^{\omega}_{<:}$ の拡張である。

32.3 節で見た、純粋な二階の有界量化の欠点を解決しようとする提案は、数多く存在する。この章で見た高階の有界量化と多相レコード更新の二つ以外にも、他の形式の多相レコード更新（Cardelli and Mitchell [87]; Cardelli [80]; Fisher and Mitchell [153]; Poll [388]）や、再帰型の**構造的展開**（Abadi and Cardelli [4]）、**正の部分型付け**（Hofmann and Pierce [214]）、存在型の**多相再パッケージ化**（Pierce [367]）、**型分解子**（Hofmann and Pierce [216]）がある。

列変数多相に基づく別の方法は Wand [487; 488; 490]、Rémy [400; 399; 403]、Vouillon [477; 478] らによって研究され、OCaml のオブジェクト指向機能の基礎となった（Rémy and Vouillon [406]）。

> 「初めから始めよ」と王様が、重々しい口調で言いました。「最後にいたるまで読み進み、しかるのちやめよ」[†3]
>
> —*Lewis Carroll*

†3（訳注）福島正実 訳『不思議の国のアリス』（角川書店、1975 年）より。



付録

演習の解答

解答 3.2.4. $|S_{i+1}| = |S_i|^3 + |S_i| \times 3 + 3$ であり、かつ $|S_0| = 0$。よって $|S_3| = 59439$。

解答 3.2.5. 帰納法による単純な証明でうまくいく。$i = 0$ のとき、証明すべきことは何もない。次に、ある $j \geq 0$ について $i = j + 1$ と $S_j \subseteq S_i$ を仮定して、$S_i \subseteq S_{i+1}$ を示さなければならない。つまり、任意の項 $\mathtt{t} \in S_i$ について、$\mathtt{t} \in S_{i+1}$ を示す。$\mathtt{t} \in S_i$ を仮定すると、三つの集合の和集合である S_i の定義より、t は次の三つの形のどれかでなければならないことがわかる。

(1) $\mathtt{t} \in \{\mathtt{true}, \mathtt{false}, 0\}$。この場合、$S_{i+1}$ の定義より明らかに $\mathtt{t} \in S_{i+1}$ である。
(2) $\mathtt{t} = \mathtt{succ}\ \mathtt{t}_1, \mathtt{pred}\ \mathtt{t}_1, \mathtt{iszero}\ \mathtt{t}_1$（ただし $\mathtt{t}_1 \in S_j$）。$S_j \subseteq S_i$ であるため、帰納法の仮定より $\mathtt{t}_1 \in S_i$ であり、よって S_{i+1} の定義より $\mathtt{t} \in S_{i+1}$。
(3) $\mathtt{t} = \mathtt{if}\ \mathtt{t}_1\ \mathtt{then}\ \mathtt{t}_2\ \mathtt{else}\ \mathtt{t}_3$（ただし $\mathtt{t}_1, \mathtt{t}_2, \mathtt{t}_3 \in S_j$）。再び、帰納法の仮定より $S_j \subseteq S_i$ であるため、S_{i+1} の定義より $\mathtt{t} \in S_{i+1}$。

解答 3.3.4. （深さに関する帰納法の原理についてのみ議論する。残りの議論は同様である。）項 s それぞれについて、もし s より小さな深さを持つ任意の r について $P(\mathtt{r})$ ならば、$P(\mathtt{s})$ である、という前提が与えられている。このとき、$P(\mathtt{s})$ がすべての s で成り立つことを示さなければならない。自然数の上の述語 Q を次の通り定義する。

$$Q(n) \quad = \quad \forall \mathtt{s}.\ depth(\mathtt{s}) = n \Rightarrow P(\mathtt{s})$$

そして自然数上の帰納法（公理 2.4.2）を用いて、すべての n について $Q(n)$ が成り立つことを示せばよい。

解答 3.5.5. P を評価関係式の導出に関する述語とする。

各導出 $\mathcal{D}$ に対して、
　すべての直接の部分導出 C に対して $P(C)$ が成り立つとき、
　$P(\mathcal{D})$ が証明できる
ならば、すべての $\mathcal{D}$ に対して $P(\mathcal{D})$ が成り立つ。

解答 3.5.10.

$$\frac{\mathtt{t} \longrightarrow \mathtt{t}'}{\mathtt{t} \longrightarrow^* \mathtt{t}'}$$

$$\mathtt{t} \longrightarrow^* \mathtt{t}$$

$$\frac{\mathtt{t} \longrightarrow^* \mathtt{t}' \qquad \mathtt{t}' \longrightarrow^* \mathtt{t}''}{\mathtt{t} \longrightarrow^* \mathtt{t}''}$$

解答 3.5.13. (1) 定理 3.5.4 と定理 3.5.11 は成り立たなくなる。定理 3.5.7 と定理 3.5.8 と定理 3.5.12 はこれまで通り妥当である。(2) 今回は定理 3.5.4 だけが成り立たなくなり、他はこれまで通り妥当である。(2) において興味深いのは、1 ステップ評価がこの規則のせいで非決定的になっても、多ステップ評価の最後の結果は決定的なままであるということである。すべての道はローマに至る。実際このことを厳密に証明するのは、以前ほど自明ではないものの、難しくはない。主な観察は、1 ステップ評価関係はいわゆる**ダイヤモンド性**を持つということである。

補題 A.1.［ダイヤモンド性］ もし $r \longrightarrow s$ かつ $r \longrightarrow t$（ただし $s \neq t$）ならば、ある項 u が存在して $s \longrightarrow u$ かつ $t \longrightarrow u$[†1] □

［証明］ 評価規則から明らかに、この状況が発生しうるのは r が `if` r_1 `then` r_2 `else` r_3 の形をしているときだけである。$r \longrightarrow s$ と $r \longrightarrow t$ の導出の**二つ組**に関する帰納法によって進む。両方の導出における最後の規則に関する場合分けを行う。

場合 i

$r \longrightarrow s$ は E-IfTrue によるもの、$r \longrightarrow t$ は E-Funny2 によるものとする。このとき、これらの規則の形から、$s = r_2$ であることと、$t =$ `if true then` r'_2 `else` r_3 であることがわかる（ただし $r_2 \longrightarrow r'_2$）。ここで $u = r'_2$ とすると、所要の通りとなる。なぜなら、$s \longrightarrow r'_2$ であることはわかっていて、$t \longrightarrow r'_2$ は E-IfTrue からわかるからである。

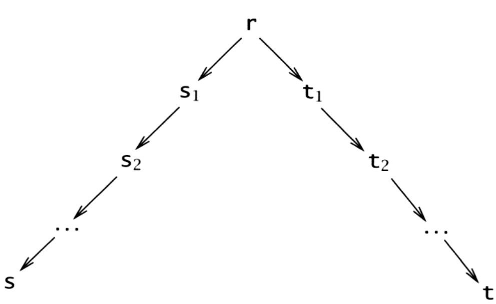
場合 ii

どちらも E-If であるとする。E-If の形から、s は `if` r'_1 `then` r_2 `else` r_3 という形であり、さらに t は `if` r''_1 `then` r_2 `else` r_3 という形であるとわかる（ただし $r_1 \longrightarrow r'_1$ かつ $r_1 \longrightarrow r''_1$）。しかしこのとき、帰納法の仮定により、ある項 r'''_1 が存在して $r'_1 \longrightarrow r'''_1$ かつ $r''_1 \longrightarrow r'''_1$ である。$u =$ `if` r'''_1 `then` r_2 `else` r_3 とし、E-If によって $s \longrightarrow u$ かつ $t \longrightarrow u$ となることを確認すれば、この場合の議論は完了である。

その他の場合

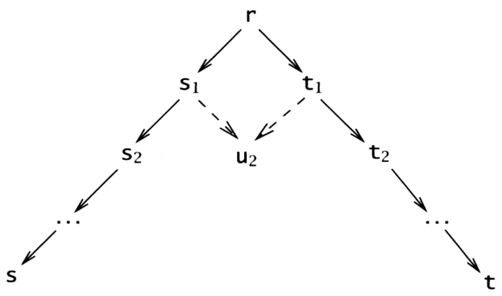
同様に示せる。 □

結果の一意性の証明は、単純な「図の補助」によって得られる。$r \longrightarrow^* s$ かつ $r \longrightarrow^* t$ とする。

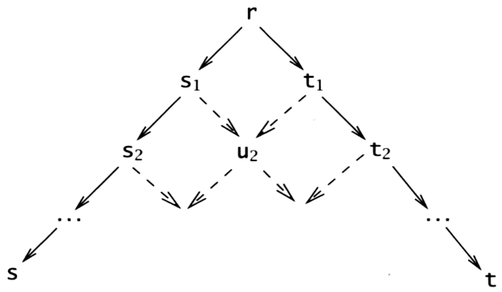


ここで補題 A.1 を用いて、s_1 と t_1 を「引き合わせる」ことができる。

†1 (訳注：原著正誤表より) この言明のダイヤモンド性は成立しない。$r \longrightarrow s$ が E-IfFalse により、$r \longrightarrow t$ が E-Funny2 によるとき、$s \longrightarrow u$ かつ $t \longrightarrow u$ なる u が存在するのではなく、$t \longrightarrow s$ となる。以下のような言明とすべきである。「もし $r \longrightarrow s$ かつ $r \longrightarrow t$（ただし $s \neq t$）ならば、$s \longrightarrow t$ であるか、$t \longrightarrow s$ であるか、さもなくばある項 u が存在して $s \longrightarrow u$ かつ $t \longrightarrow u$。」以下の議論もこの修正に合わせて若干読み替える必要がある。



次に s_2 と u_2 を引き合わせ、さらに u_2 と t_2 も引き合わせる。



これを s と t を引き合わせるまで繰り返し、個々の 1 ステップのダイヤモンドから完全な大ダイヤモンドを構築する。

これにより、r は v へも w へも評価できるとき、v と w は**同一でなければならない**といえる。（v と w は共に正規形であるので、それらが同じものに評価できる唯一の方法は、元から同じものである場合だけである。）

解答 3.5.14. t の構造に関する帰納法。

t が値の場合

すべての値は正規形であるので、この場合は起きえない。

$t = \mathsf{succ}\ t_1$ の場合

評価規則を見ると、$t \longrightarrow t'$ かつ $t \longrightarrow t''$ を導出するのに用いることができる規則は E-Succ だけであるとわかる（他の規則はすべて左辺の最外の構築子が succ ではない）。よって結論が $t_1 \longrightarrow t_1'$ と $t_1 \longrightarrow t_1''$ の形であるような二つの部分導出が存在しなければならない。帰納法の仮定により（t_1 は t の部分項であるので適用可能）、$t_1' = t_1''$ を得る。しかしこのとき、$\mathsf{succ}\ t_1' = \mathsf{succ}\ t_1''$ である。これは所要の通りである。

$t = \mathsf{pred}\ t_1$ の場合

ここでは、t を t' や t'' に簡約するのに用いることができる規則が三つある（E-Pred と E-PredZero と E-PredSucc）。しかしながら、これらの規則に重なりはない。t がいずれかの規則の左辺に適合するなら、他の規則の左辺には全く適合**しない**。（例えば t が E-Pred に適合するなら、t_1 は値ではない。つまり、0 でも succ v でもない。）このことから、$t \longrightarrow t'$ と $t \longrightarrow t''$ の導出に用いられる規則は**同一**であるとわかる。この規則が E-Pred であれば、$t = \mathsf{succ}\ t_1$ の場合と同様に帰納法の仮定を用いる。E-PredZero か E-PredSucc であれば、結論は直ちに得られる。

その他の場合

同様に示せる。

解答 3.5.16. wrong で拡張された項の集合（wrong を部分項として含む項も含む）の上を動くメタ変数として t を用い、wrong を含まない "good" な項の集合の上を動くメタ変数として g を用いることにしよう。wrong の遷移で拡張された新しい評価関係を $t \xrightarrow{w} t'$ と書く。また、元の評価を $g \xrightarrow{o} g'$ と書く（original の o）。さて、二つの取り扱いが一致するという主張は、次のように正確に形式化できる。元の意味論では評価が行き詰まり状態になってしまう任意の（元の）項は、新たな意味論では wrong に評価され、その逆も成り立つ[†2]。形式的に記すと、

命題 A.2. すべての元の項 g について、($g \xrightarrow{o}{}^* g'$、ただし g' は行き詰まり状態) のとき、かつそのときに限り、($g \xrightarrow{w}{}^*$ wrong) □

これを証明するには、幾つか段階を踏む。まずは、追加した新しい遷移が演習 3.5.14 の定理を壊さないことを述べる。

補題 A.3. 拡張された評価関係は決定的である。 □

これはつまり、g が元の意味論で g' へ 1 ステップで遷移できるときは必ず、拡張された意味論でも g' へステップを持ち、さらに g' は新しい意味論において g から遷移できる**唯一の**項である、ということを意味する。

次に、元の意味論で行き詰まり状態である項は、拡張された意味論では常に wrong に評価されることを示す。

補題 A.4. g が行き詰まり状態ならば、$g \xrightarrow{w}{}^*$ wrong □

[証明] g の構造に関する帰納法による。

g = true または false または 0 の場合

起きえない（g は行き詰まり状態であると仮定したので）。

g = if g_1 then g_2 else g_3 の場合

g が行き詰まり状態であるから、g_1 は正規形でなければならない（さもなくば E-If が適用できる）。明らかに g_1 は true または false ではありえない（さもなくば、E-IfTrue か E-IfFalse のいずれかが適用できて g は行き詰まり状態ではないことになる）。残りの場合を考える。

g_1 = if g_{11} then g_{12} else g_{13} の場合

g_1 は正規形であるが、しかし明らかに値ではないから、これは行き詰まり状態である。帰納法の仮定により、$g_1 \xrightarrow{w}{}^*$ wrong とわかる。ここから、$g \xrightarrow{w}{}^*$ if wrong then g_2 else g_3 の導出を構築することができる。規則 E-If-Wrong のインスタンスを最後に追加することで、所要の通り $g \xrightarrow{w}{}^*$ wrong が得られる。

g_1 = succ g_{11} の場合

g_{11} が数値である場合、g_1 は badbool であり、規則 E-If-Wrong から直ちに $g \xrightarrow{w}$ wrong が得られる。そうでなければ、定義から g_1 は行き詰まり状態であり、帰納法の仮定から $g_1 \xrightarrow{w}{}^*$ wrong といえる。この導出から、$g \xrightarrow{w}{}^*$ if wrong then g_2 else g_3 の導出を構築する。規則 E-If-Wrong のインスタンスを最後に追加することで、$g \xrightarrow{w}{}^*$ wrong を得られる。

その他の場合

同様。

[†2]（訳注：原著正誤表より）二つの取り扱いが一致するという主張を完全に捉えるには、次の（ほぼ明らかな）性質も示すべきである。すべての元の項 g について、$g \xrightarrow{o}{}^* v$ のとき、かつそのときに限り、$g \xrightarrow{w}{}^* v$ 。

$\mathtt{g} = \mathtt{succ}\ \mathtt{g}_1$ の場合

$\mathtt{g}$ が行き詰まり状態であるから、(値の定義より) $\mathtt{g}_1$ は正規形であるが数値ではないとわかる。二つの可能性がある。$\mathtt{g}_1$ が true か false である可能性か、または $\mathtt{g}_1$ 自体が値ではない、すなわち行き詰まり状態である可能性である。一つ目の可能性では、規則 E-SUCC-WRONG によって直ちに $\mathtt{g} \xrightarrow{w} \mathtt{wrong}$ が得られる。二つ目の可能性では、帰納法の仮定により $\mathtt{g}_1 \xrightarrow{w}{}^{*} \mathtt{wrong}$ であり、あとの議論は前回と同様である。

その他の場合

同様に示せる。 □

補題 A.3 と補題 A.4 を組み合わせて、命題 A.2 の十分性 ($\Longrightarrow$) の証明が得られる。もう一方に対しては、拡張された意味論において「今まさに wrong になろうとしている」項が元の意味論では行き詰まり状態であることを証明する必要がある。

補題 A.5. 拡張された意味論において $\mathtt{g} \xrightarrow{w} \mathtt{t}$ であり $\mathtt{t}$ が wrong を部分項として含むならば、$\mathtt{g}$ は元の意味論において行き詰まり状態である。 □

[証明] (拡張された) 評価導出に関する単純な帰納法。 □

これと補題 A.3 を組み合わせて、命題 A.2 の必要性 ($\Longleftarrow$) の証明が得られる。以上で完了である。

解答 3.5.17. 十分性と必要性をそれぞれ命題 A.7 と命題 A.9 で証明する。どちらの場合も、(小ステップによる) 多ステップ評価に関する有用な性質を幾つか証明する技術的な補題から取り掛かる。

補題 A.6. $\mathtt{t}_1 \longrightarrow^* \mathtt{t}_1'$ ならば $\mathtt{if}\ \mathtt{t}_1\ \mathtt{then}\ \mathtt{t}_2\ \mathtt{else}\ \mathtt{t}_3 \longrightarrow^* \mathtt{if}\ \mathtt{t}_1'\ \mathtt{then}\ \mathtt{t}_2\ \mathtt{else}\ \mathtt{t}_3$ である (他の項構築子についても同様)。 □

[証明] 簡単な帰納法。 □

命題 A.7. $\mathtt{t} \Downarrow \mathtt{v}$ ならば $\mathtt{t} \longrightarrow^* \mathtt{v}$ □

[証明] $\mathtt{t} \Downarrow \mathtt{v}$ の導出に関する帰納法による。最後に用いられる規則について場合分けを行う。

B-VALUE の場合 $\mathtt{t} = \mathtt{v}$

直ちに得られる。

B-IFTRUE の場合 $\mathtt{t} = \mathtt{if}\ \mathtt{t}_1\ \mathtt{then}\ \mathtt{t}_2\ \mathtt{else}\ \mathtt{t}_3$
$\mathtt{t}_1 \Downarrow \mathtt{true}$
$\mathtt{t}_2 \Downarrow \mathtt{v}$

帰納法の仮定により $\mathtt{t}_1 \longrightarrow^* \mathtt{true}$。補題 A.6 より、

$$\mathtt{if}\ \mathtt{t}_1\ \mathtt{then}\ \mathtt{t}_2\ \mathtt{else}\ \mathtt{t}_3 \longrightarrow^* \mathtt{if}\ \mathtt{true}\ \mathtt{then}\ \mathtt{t}_2\ \mathtt{else}\ \mathtt{t}_3.$$

E-IFTRUE により、$\mathtt{if}\ \mathtt{true}\ \mathtt{then}\ \mathtt{t}_2\ \mathtt{else}\ \mathtt{t}_3 \longrightarrow \mathtt{t}_2$。帰納法の仮定により、$\mathtt{t}_2 \longrightarrow^* \mathtt{v}$。ここから $\longrightarrow^*$ の推移律によって結論が得られる。

その他の場合

同様に示せる。 □

補題 A.8.　もし

$$\mathtt{if}\ \mathtt{t}_1\ \mathtt{then}\ \mathtt{t}_2\ \mathtt{else}\ \mathtt{t}_3 \longrightarrow^* \mathtt{v}$$

ならば、$\mathtt{t}_1 \longrightarrow^* \mathtt{true}$ かつ $\mathtt{t}_2 \longrightarrow^* \mathtt{v}$ であるか、$\mathtt{t}_1 \longrightarrow^* \mathtt{false}$ かつ $\mathtt{t}_3 \longrightarrow^* \mathtt{v}$ である。さらに、$\mathtt{t}_1$ と $\mathtt{t}_2$ または $\mathtt{t}_3$ の評価の列は、与えられた評価の列より厳密に短い。（他の項構築子についても同様である。）　□

［証明］　与えられた評価の列の長さに関する帰納法による。条件式は値ではないので、評価を少なくとも 1 ステップ進められるはずである。このステップの最後に用いられる規則について場合分けを行う（IF の規則のどれかであることに注意されたい）。

E-IF の場合　$\mathtt{if}\ \mathtt{t}_1\ \mathtt{then}\ \mathtt{t}_2\ \mathtt{else}\ \mathtt{t}_3 \longrightarrow \mathtt{if}\ \mathtt{t}'_1\ \mathtt{then}\ \mathtt{t}_2\ \mathtt{else}\ \mathtt{t}_3 \longrightarrow^* \mathtt{v}$

$\mathtt{t}_1 \longrightarrow \mathtt{t}'_1$

帰納法の仮定により、$\mathtt{t}'_1 \longrightarrow^* \mathtt{true}$ かつ $\mathtt{t}_2 \longrightarrow^* \mathtt{v}$ であるか、$\mathtt{t}'_1 \longrightarrow^* \mathtt{false}$ かつ $\mathtt{t}_3 \longrightarrow^* \mathtt{v}$ である。$\mathtt{t}'_1 \longrightarrow^* \mathtt{true}$ か $\mathtt{t}'_1 \longrightarrow^* \mathtt{false}$ の導出に $\mathtt{t}_1 \longrightarrow \mathtt{t}'_1$ を最初のステップとして追加すると所望の結論が得られる。この評価の列が元の列より短いことを確認するのは簡単である。

E-IFTRUE の場合　$\mathtt{if}\ \mathtt{true}\ \mathtt{then}\ \mathtt{t}_2\ \mathtt{else}\ \mathtt{t}_3 \longrightarrow \mathtt{t}_2 \longrightarrow^* \mathtt{v}$

直ちに得られる。

E-IFFALSE の場合　$\mathtt{if}\ \mathtt{false}\ \mathtt{then}\ \mathtt{t}_2\ \mathtt{else}\ \mathtt{t}_3 \longrightarrow \mathtt{t}_3 \longrightarrow^* \mathtt{v}$

直ちに得られる。　□

命題 A.9.　$\mathtt{t} \longrightarrow^* \mathtt{v}$ ならば $\mathtt{t} \Downarrow \mathtt{v}$　□

［証明］　与えられた $\mathtt{t} \longrightarrow^* \mathtt{v}$ の導出における小ステップ評価のステップの数に関する帰納法による。

$\mathtt{t} \longrightarrow^* \mathtt{v}$ が 0 ステップならば、$\mathtt{t} = \mathtt{v}$ であり B-VALUE によって結論が得られる。それ以外の場合、$\mathtt{t}$ の形に関する場合分けを行う。

$\mathtt{t} = \mathtt{if}\ \mathtt{t}_1\ \mathtt{then}\ \mathtt{t}_2\ \mathtt{else}\ \mathtt{t}_3$ の場合

補題 A.8 より、(1) $\mathtt{t}_1 \longrightarrow^* \mathtt{true}$ かつ $\mathtt{t}_2 \longrightarrow^* \mathtt{v}$、または、(2) $\mathtt{t}_1 \longrightarrow^* \mathtt{false}$ かつ $\mathtt{t}_3 \longrightarrow^* \mathtt{v}$。どちらの場合も同様の議論を行うので、(1) の場合だったとしよう。さらに補題 A.8 から、$\mathtt{t}_1 \longrightarrow^* \mathtt{true}$ と $\mathtt{t}_2 \longrightarrow^* \mathtt{v}$ の評価のステップは、$\mathtt{t}$ の評価のステップより短いとわかる。よって帰納法の仮定が適用でき、$\mathtt{t}_1 \Downarrow \mathtt{true}$ と $\mathtt{t}_2 \Downarrow \mathtt{v}$ が得られる。これらから、規則 B-IFTRUE を用いて $\mathtt{t} \Downarrow \mathtt{v}$ を導出できる。

その他の項構築子の場合

同様に示せる。　□

解答 4.2.1.　`eval` は自分自身を呼ぶたびに、`try` ハンドラを呼び出しスタックに追加する。評価のステップごとに一回の再帰呼び出しがあるため、スタックはいつかオーバーフローしてしまう。本質的には、`eval` の再帰呼び出しを `try` で囲むと（そうは見えないものの）末尾呼び出しではなくなるということである。より良い（しかし読みづらい）`eval` は次のようになる。

```
let rec eval t =
  let t'opt = try Some(eval1 t) with NoRuleApplies → None in
  match t'opt with
    Some(t') → eval t'
  | None → t
```

解答 5.2.1.

```
or = λb. λc. b tru c;
not = λb. b fls tru;
```

解答 5.2.2.

```
scc2 = λn. λs. λz. n s (s z);
```

解答 5.2.3.

```
times2 = λm. λn. λs. λz. m (n s) z;
```

もしくはもっと簡潔に、

```
times3 = λm. λn. λs. m (n s);
```

解答 5.2.4. この演習も、解法は複数ある。

```
power1 = λn. λm. m (times n) c₁;
power2 = λn. λm. m n;
```

解答 5.2.5.

```
subtract1 = λm. λn. n prd m;
```

解答 5.2.6. prd c_n を評価するには、$O(n)$ ステップかかる。なぜなら、prd は、数の二つ組 n 個の列を作り、その列の最後の二つ組の最初の要素を選ぶからである。

解答 5.2.7. 以下に簡単なものを示す。

```
equal = λm. λn.
          and (iszro (m prd n))
              (iszro (n prd m));
```

解答 5.2.8. 著者が想定していた解答は次の通りである。

```
nil = λc. λn. n;
cons = λh. λt. λc. λn. c h (t c n);
head = λl. l (λh.λt.h) fls;
tail = λl.
         fst (l (λx. λp. pair (snd p) (cons x (snd p)))
                (pair nil nil));
isnil = λl. l (λh.λt.fls) tru;
```

次のようなかなり異なる方法もある。

```
nil = pair tru tru;
cons = λh. λt. pair fls (pair h t);
head = λz. fst (snd z);
tail = λz. snd (snd z);
isnil = fst;
```

解答 5.2.9. test ではなく if を使ったのは、条件式の両方の分岐が常に評価されてしまうことを避けるためである。両方を評価すると factorial は発散してしまう。test を使う場合にこの発散を避けるためには、両方の分岐をダミーのラムダ抽象で包み込んで保護する必要がある。ラムダ抽象は値なので、値呼び評価戦略がその中まで見に行くことはない。その代わり、test にそのラムダ抽象二つをそのまま渡す。test はどちらかの値を選択し、返す。そこで、test 式全体をダミー引

数（例えば c_0）に適用し、選ばれた分岐を評価するよう強制する。

```
ff = λf. λn.
       test
         (iszro n) (λx. c₁) (λx. (times n (f (prd n)))) c₀;
factorial = fix ff;
equal c₆ (factorial c₃);
▸ (λx. λy. x)
```

解答 5.2.10. これを行う再帰関数は次の通りである。

```
cn = λf. λm. if iszero m then c₀ else scc (f (pred m));
churchnat = fix cn;
```

これが動作することの簡単な検査は次の通り。

```
equal (churchnat 4) c₄;
▸ (λx. λy. x)
```

解答 5.2.11.

```
ff = λf. λl.
       test (isnil l)
          (λx. c₀) (λx. (plus (head l) (f (tail l)))) c₀;
sumlist = fix ff;

l = cons c₂ (cons c₃ (cons c₄ nil));
equal (sumlist l) c₉;
▸ (λx. λy. x)
```

リストの和を取る関数は、当然、`fix` を使わなくても書ける。

```
sumlist' = λl. l plus c₀;
equal (sumlist' l) c₉;
▸ (λx. λy. y)
```

解答 5.3.3. t のサイズに関する帰納法による。t より小さな項に対して、所望の性質が満たされていると仮定し、t 自体に対しても所望の性質が満たされることを証明しなければならない。もしも証明に成功すれば、すべての t について性質が満たされることを結論付けられる。三つの場合を考える。

$t = x$ の場合
直ちに得られる。$|FV(t)| = |\{x\}| = 1 = size(t)$

$t = \lambda x.t_1$ の場合
帰納法の仮定から、$|FV(t_1)| \le size(t_1)$。よって、以下のように計算できる。$|FV(t)| = |FV(t_1) \setminus \{x\}| \le |FV(t_1)| \le size(t_1) < size(t)$

$t = t_1\ t_2$ の場合
帰納法の仮定から、$|FV(t_1)| \le size(t_1)$ かつ $|FV(t_2)| \le size(t_2)$。よって、以下のように計算できる。$|FV(t)| = |FV(t_1) \cup FV(t_2)| \le |FV(t_1)| + |FV(t_2)| \le size(t_1) + size(t_2) < size(t)$

解答 5.3.6. （非決定的な）完全ベータ簡約に対しては、規則は以下のようになる。

$$\frac{t_1 \longrightarrow t_1'}{t_1\ t_2 \longrightarrow t_1'\ t_2} \qquad \text{(E-APP1)}$$

$$\frac{\mathtt{t_2} \longrightarrow \mathtt{t_2'}}{\mathtt{t_1\ t_2} \longrightarrow \mathtt{t_1\ t_2'}} \qquad \text{(E-APP2)}$$

$$\frac{\mathtt{t_1} \longrightarrow \mathtt{t_1'}}{\lambda\mathtt{x.t_1} \longrightarrow \lambda\mathtt{x.t_1'}} \qquad \text{(E-ABS)}$$

$$(\lambda\mathtt{x.t_{12}})\ \mathtt{t_2} \longrightarrow [\mathtt{x} \mapsto \mathtt{t_2}]\mathtt{t_{12}} \qquad \text{(E-APPABS)}$$

(値の構文要素は使っていないことに注意されたい。)

正規順序戦略に対しては、以下が規則を書き下す一つの方法である。

$$\frac{\mathtt{na_1} \longrightarrow \mathtt{t_1'}}{\mathtt{na_1\ t_2} \longrightarrow \mathtt{t_1'\ t_2}} \qquad \text{(E-APP1)}$$

$$\frac{\mathtt{t_2} \longrightarrow \mathtt{t_2'}}{\mathtt{nanf_1\ t_2} \longrightarrow \mathtt{nanf_1\ t_2'}} \qquad \text{(E-APP2)}$$

$$\frac{\mathtt{t_1} \longrightarrow \mathtt{t_1'}}{\lambda\mathtt{x.t_1} \longrightarrow \lambda\mathtt{x.t_1'}} \qquad \text{(E-ABS)}$$

$$(\lambda\mathtt{x.t_{12}})\ \mathtt{t_2} \longrightarrow [\mathtt{x} \mapsto \mathtt{t_2}]\mathtt{t_{12}} \qquad \text{(E-APPABS)}$$

ただし、正規形、非抽象正規形、非抽象の構文要素は、以下のように定義される。

nf ::= 正規形：

  λx.nf

  nanf

nanf ::= 非抽象正規形：

  x

  nanf nf

na ::= 非抽象：

  x

  $\mathtt{t_1\ t_2}$

(この定義は、他と比べると少々不格好である。正規順序簡約は通常、「完全ベータ簡約のようなものだが、最も左で、最も外側の簡約基が常に最初に選ばれるという点だけが異なる」という説明だけで定義される。)

遅延戦略は、値を任意のラムダ抽象として定義する。つまり、値呼びと同じである。評価規則は次の通りである。

$$\frac{\mathtt{t_1} \longrightarrow \mathtt{t_1'}}{\mathtt{t_1\ t_2} \longrightarrow \mathtt{t_1'\ t_2}} \qquad \text{(E-APP1)}$$

$$(\lambda\mathtt{x.t_{12}})\ \mathtt{t_2} \longrightarrow [\mathtt{x} \mapsto \mathtt{t_2}]\mathtt{t_{12}} \qquad \text{(E-APPABS)}$$

解答 5.3.8.

$$\lambda\mathtt{x.t} \Downarrow \lambda\mathtt{x.t}$$

$$\frac{\mathtt{t}_1 \Downarrow \lambda\mathtt{x.t}_{12} \qquad \mathtt{t}_2 \Downarrow \mathtt{v}_2 \qquad [\mathtt{x} \mapsto \mathtt{v}_2]\mathtt{t}_{12} \Downarrow \mathtt{v}}{\mathtt{t}_1\ \mathtt{t}_2 \Downarrow \mathtt{v}}$$

解答 6.1.1.

```
c0 = λ. λ. 0;
c2 = λ. λ. 1 (1 0);
plus = λ. λ. λ. λ. 3 1 (2 1 0);
fix = λ. (λ. 1 (λ. (1 1) 0)) (λ. 1 (λ. (1 1) 0));
foo = (λ. (λ. 0)) (λ. 0);
```

解答 6.1.5. この二つの関数は以下のように定義できる。

$removenames_\Gamma(\mathtt{x})$	=	Γ において最も右にある x のインデックス
$removenames_\Gamma(\lambda\mathtt{x.t}_1)$	=	$\lambda.\ removenames_{\Gamma,\mathtt{x}}(\mathtt{t}_1)$
$removenames_\Gamma(\mathtt{t}_1\ \mathtt{t}_2)$	=	$removenames_\Gamma(\mathtt{t}_1)\ removenames_\Gamma(\mathtt{t}_2)$
$restorenames_\Gamma(\mathtt{k})$	=	Γ における k 番目の名前
$restorenames_\Gamma(\lambda.\mathtt{t}_1)$	=	$\lambda\mathtt{x}.\ restorenames_{\Gamma,\mathtt{x}}(\mathtt{t}_1)$ ただし x は $dom(\Gamma)$ に含まれないような最初の名前
$restorenames_\Gamma(\mathtt{t}_1\ \mathtt{t}_2)$	=	$restorenames_\Gamma(\mathtt{t}_1)\ restorenames_\Gamma(\mathtt{t}_2)$

removenames と *restorenames* について所要の性質は、項についての単純な構造的帰納法によって証明できる。

解答 6.2.2.

(1) λ. λ. 1 (0 4)
(2) λ. 0 3 (λ. 0 1 4)

解答 6.2.5.

[0 ↦ 1] (0 (λ.λ.2))	=	1 (λ. λ. 3)
	すなわち	a (λx. λy. a)
[0 ↦ 1 (λ.2)] (0 (λ.1))	=	(1 (λ.2)) (λ.(2 (λ.3)))
	すなわち	(a (λz.a)) (λx.(a (λz.a)))
[0 ↦ 1] (λ. 0 2)	=	λ. 0 2
	すなわち	λb. b a
[0 ↦ 1] (λ. 1 0)	=	λ. 2 0
	すなわち	λa′. a a′

解答 6.2.8. Γ が名前付け文脈のとき、右から数えた Γ の中の x のインデックスを $\Gamma(\mathtt{x})$ と書く。すると、我々の欲しい性質は、以下のように書ける。

$$removenames_\Gamma([\mathtt{x} \mapsto \mathtt{s}]\mathtt{t}) \ = \ [\Gamma(\mathtt{x}) \mapsto removenames_\Gamma(\mathtt{s})](removenames_\Gamma(\mathtt{t}))$$

証明は、定義 5.3.5 と定義 6.2.4 を用いた t についての帰納法と、簡単な計算、*removenames* や項についての基本的な演算についての簡単な補題を用いて進む。慣習 5.3.4 が抽象化の場合について重要な役割を演じる。

解答 6.3.1. インデックスが負になりうる唯一の可能性は、0 の番号の変数が、シフトする項のどこかに実際に現れるときである。しかし、これは起きえない。なぜなら、変数 0 への代入を行ったばかりだから（であり、変数 0 に対して代入した項は既に +1 シフトされており、変数番号 0 のいかなるインスタンスも明らかに含まないから）である。

解答 6.3.2. インデックスとレベルを用いた表現の等価性の証明は Lescanne and Rouyer-Degli [285] にある。de Bruijn レベルについては、de Bruijn [134] や Filinski [148] の Section 5.2 でも議論されている。

解答 8.3.5. できない。この規則を取り除くと進行の性質が壊れてしまう。0 の前者値を定義するのがどうしても嫌なら、他の方法で扱う必要がある。例えば、0 の前者値を求めようとしたら例外を投げることにしたり（第 14 章）、おそらく交差型（15.7 節）や依存型（30.5 節）を用いて `pred` の型を詳細化し、これが厳密に正の数にしか適用できないことをはっきりさせたりすればよい。

解答 8.3.6. 反例を挙げる。`(if false then true else 0)` という項は正しく型付けされないが、評価すれば正しく型付けされた項 `0` になる。

解答 8.3.7. 大ステップ意味論での型保存の性質は、小ステップ意味論に与えたものと同様である。すなわち、正しく型付けされた項は何らかの最終的な値に評価され、その値は元の項と同じ型を持つということである。その証明も、小ステップ意味論と同様である。一方、進行の性質は、ずっと強い主張を行うことになる。すなわち、正しく型付けされた項はすべて最終的な値に評価される、ということになる。つまり、正しく型付けされた項においては評価は常に停止する、と述べている。算術式に対してはたまたまこれは成立するが、より興味深い言語（例えば一般的再帰を含む言語。11.11 節参照）においてはしばしば成立しない。そのような言語においては、進行の性質はそもそも**成立しない**。事実上、エラー状態に到達することと停止しないことの違いを述べる方法はない。これは言語理論家が一般的に小ステップを好む理由の一つである。

別の方法として、演習 8.3.8 で見るように、`wrong` への遷移を明示的に持つ大ステップ意味論を与える方法がある。この方法は、例えば Abadi と Cardelli が、彼らのオブジェクト計算の操作的意味論のために用いている（Abadi and Cardelli [4] p.87)。

解答 8.3.8. 拡張された意味論においては行き詰まり状態というものは存在しない。値でない項はすべて、普通に他の項に評価されるか、または明示的に `wrong` に評価される（これは当然**証明**しなければならない)。よって進行の性質は自明である。一方、主部簡約定理はもう少し意味を持つ。`wrong` は型を持たないから、正しく型付けされた項が正しく型付けされた他の項にのみ評価されるということは、正しく型付けされた項が `wrong` にステップすることはありえないということである。実際、以前の進行定理の証明は、新しい保存の証明の一部となる。

解答 9.2.1. なぜなら、型式の集合が空だからである（型の構文に基底ケースがない)。

解答 9.2.3. そのような文脈の一つは

$$\Gamma = \texttt{f:Bool}\to\texttt{Bool}\to\texttt{Bool},\ \texttt{x:Bool},\ \texttt{y:Bool}$$

である。一般に、任意の型 S と T について、次の形の文脈

$$\Gamma = \texttt{f:S}\to\texttt{T}\to\texttt{Bool},\ \texttt{x:S},\ \texttt{y:T}$$

で型付けできる。この種の推論が、第 22 章で説明する**型再構築**アルゴリズムの中心である。

解答 9.3.2. 矛盾を導く目的で、仮に x x が型 T をまさに持っているとする。すると、逆転補題より、左の項（x）は $T_1 \to T_2$ を持ち、右の項（同様に x）は型 T_1 を持たなければならない。逆転補題の変数のケースを使えば、$x : T_1 \to T_2$ と $x : T_1$ の両方が Γ 中の仮定より導けなければならないことがわかる。Γ 中の x の束縛は、ただ一つしかありえないため、これは $T_1 \to T_2 = T_1$ を意味する。しかしこれはありえない。すべての型は有限のサイズを持つので、型の一部分に自分自身が現れることはできない。よって矛盾。

もし型が無限に大きくてもよいならば、等式 $T_1 \to T_2 = T_1$ の解を構築**できうる**ことに注意されたい。この点については第 20 章で詳細に述べる。

解答 9.3.3. $\Gamma \vdash t : S$ かつ $\Gamma \vdash t : T$ を仮定する。$\Gamma \vdash t : T$ の導出に関する帰納法により、$S = T$ を示す。

T-VAR の場合　$t = x$
　ただし $x : T \in \Gamma$

逆転補題（補題 9.3.1）の (1) より、$\Gamma \vdash t : S$ の任意の導出における最後の規則も T-VAR でなければならない。よって、$S = T$。

T-ABS の場合　$t = \lambda y : T_2 . t_1$
　$T = T_2 \to T_1$
　$\Gamma, y : T_2 \vdash t_1 : T_1$

逆転補題の (2) より、$\Gamma \vdash t : S$ の任意の導出における最後の規則も T-ABS である。そしてこの導出は結論 $\Gamma, y : T_2 \vdash t_1 : S_1$（ただし $S = T_2 \to S_1$）を持つ部分導出を持つはずである。（結論 $\Gamma, y : T_2 \vdash t_1 : T_1$ を持つ部分導出に対する）帰納法の仮定により、$S_1 = T_1$ を得る。ここから直ちに $S = T$ となる。

T-APP、T-TRUE、T-FALSE、T-IF の場合
同様。

解答 9.3.9. $\Gamma \vdash t : T$ の導出に関する帰納法による。帰納法の各ステップでは、所望の性質が任意の部分導出に対して成立する（つまり、$\Gamma \vdash s : S$ かつ $s \longrightarrow s'$ のとき、$\Gamma \vdash s : S$ が現在の導出の部分導出として証明されているならば常に $\Gamma \vdash s' : S$ である）と仮定し、その導出の最後に使われている規則に関する場合分けを行う。

T-VAR の場合　$t = x$　　$x : T \in \Gamma$
起きえない（左辺に変数を持つ評価規則は存在しない）。

T-ABS の場合　$t = \lambda x : T_1 . t_2$
起きえない。

T-APP の場合　$t = t_1\ t_2$　　$\Gamma \vdash t_1 : T_{11} \to T_{12}$　　$\Gamma \vdash t_2 : T_{11}$
　$T = T_{12}$

図 9-1 の評価規則で左辺が適用を持つものを見ると、$t \longrightarrow t'$ を導出できる規則は三つあることがわかる。E-APP1 と E-APP2 と E-APPABS である。これらを個別に考える。

E-APP1 の場合　$t_1 \longrightarrow t_1'$　　$t' = t_1'\ t_2$

T-APP の場合の仮定から、元の型付け導出の部分導出で、結論が $\Gamma \vdash t_1 : T_{11} \to T_{12}$ であるものがある。この部分導出に対して帰納法の仮定を適用でき、これにより $\Gamma \vdash t_1' : T_{11} \to T_{12}$ を得られる。これと、$\Gamma \vdash t_2 : T_{11}$ であること（これも T-APP の場合の仮定にある）を組み合わせて規則 T-APP を適用することで、$\Gamma \vdash t' : T$ と結論付けられる。

E-APP2 の場合 $\mathtt{t}_1 = \mathtt{v}_1$ （つまり $\mathtt{t}_1$ は値）
$\mathtt{t}_2 \longrightarrow \mathtt{t}_2'$
$\mathtt{t}' = \mathtt{v}_1\ \mathtt{t}_2'$

同様。

E-APPABS の場合 $\mathtt{t}_1 = \lambda\mathtt{x}{:}\mathtt{T}_{11}.\ \mathtt{t}_{12}$
$\mathtt{t}_2 = \mathtt{v}_2$
$\mathtt{t}' = [\mathtt{x} \mapsto \mathtt{v}_2]\mathtt{t}_{12}$

逆転補題を用いて、$\lambda\mathtt{x}{:}\mathtt{T}_{11}.\ \mathtt{t}_{12}$ の型付け導出を分解でき、それにより $\Gamma, \mathtt{x}{:}\mathtt{T}_{11} \vdash \mathtt{t}_{12} : \mathtt{T}_{12}$ を得られる。これと代入補題（補題 9.3.8）から、$\Gamma \vdash \mathtt{t}' : \mathtt{T}_{12}$ を得る。

ブール定数と条件式の場合は定理 8.3.3 と同じである。

解答 9.3.10. (λx:Bool→Bool. λy:Bool. y) (λz:Bool. true true) という項は正しく型付けできないが、簡約すると (λy:Bool.y) となり、これは正しく型付けできる。

解答 9.4.1. T-TRUE と T-FALSE が導入規則である。T-IF が除去規則である。T-ZERO と T-SUCC は導入規則である。T-PRED と T-ISZERO は除去規則である。succ と pred が導入形式であるか除去形式であるかを判断するには少し考える必要がある。なぜなら、これらは数を作っているとも使っているともみなせるからである。重要な観察は、pred と succ が出会うと、簡約基をなすということである。iszero も同様である。

解答 11.2.1.

$$\mathtt{t}_1 \;=\; (\lambda\mathtt{x}{:}\mathtt{Unit}.\mathtt{x})\ \mathtt{unit}$$
$$\mathtt{t}_{i+1} \;=\; (\lambda\mathtt{f}{:}\mathtt{Unit}\to\mathtt{Unit}.\ \mathtt{f(f(unit))})\ (\lambda\mathtt{x}{:}\mathtt{Unit}.\mathtt{t}_i)$$

解答 11.3.2.

$$\frac{\Gamma \vdash \mathtt{t}_2 : \mathtt{T}_2}{\Gamma \vdash \lambda_{:}\mathtt{T}_1.\mathtt{t}_2 : \mathtt{T}_1\to\mathtt{T}_2} \qquad \text{(T-WILDCARD)}$$

$$(\lambda_{:}\mathtt{T}_{11}.\mathtt{t}_{12})\ \mathtt{v}_2 \longrightarrow \mathtt{t}_{12} \qquad \text{(E-WILDCARD)}$$

これらの規則が略記から導出されることの証明は、定理 11.3.1 と完全に同様である。

解答 11.4.1. 前半は簡単である。t as T $\overset{\text{def}}{=}$ (λx:T. x) t という規則を使って型指定を脱糖衣すれば、型指定の型付け規則と評価規則が共に、ラムダ抽象と適用に関する規則から直接導出できる。このことを確かめるのは単純である。型指定の評価規則を先行評価版に変更すると、指定が取り除かれるまで t の評価を遅延するような、より改良された脱糖衣を用いなくてはならない。例えば、以下の脱糖衣を用いることができる。

t as T $\overset{\text{def}}{=}$ (λx:Unit→T. x unit) (λy:Unit. t) ただし y はフレッシュ

もちろん、ここで Unit を選んでいるのは本質的ではなく、どんな型でもよい。

ここで微妙なのは、この脱糖衣が、直観的には正しいものの、定理 11.3.1 の性質を正確には**もたらさない**点である。その理由は、高水準の規則 E-ASCRIBE では型指定が消えるのに 1 ステップで済むのに対し、脱糖衣した結果では **2 ステップ**かかるためである。しかし、これは驚くべきことではない。脱糖衣は単純な形のコンパイルとみなせるし、ほとんど**すべて**の高水準機能はソース言語における原子的な簡約ごとにターゲット言語で複数の評価ステップを必要とすることが観察できる。そこで、高水準の評価ステップが低水準でのステップの**列**に対応するよう、定理 11.3.1 の要求を弱める必要がある。

$\mathtt{t} \longrightarrow_E \mathtt{t}'$ ならば $e(\mathtt{t}) \longrightarrow^*_I e(\mathtt{t}')$

最後にややこしいのは逆向き（すなわち、脱糖衣結果の項の簡約が常に元の項の簡約に「逆対応」させられるという事実）を厳密に述べるには、少しばかり注意が必要な点である。なぜなら、脱糖衣された型指定の除去には2ステップかかり、最初のステップ後の低水準な項は、元の高水準の型指定を持った項の脱糖衣結果と、型指定を除去した後の項の脱糖衣結果のいずれとも**対応しない**ためである。それでも**成り立っている**ことは、脱糖衣した項の最初の簡約は、別の高水準な項の脱糖衣結果に到達するよう、もう1ステップかけて「終わらせる」ことが必ずできるということである。形式的に表現すると、$e(\mathtt{t}) \longrightarrow_I s$ ならば $s \longrightarrow^* e(\mathtt{t}')$ かつ $\mathtt{t} \longrightarrow_E \mathtt{t}'$ である。

解答 11.5.1. 以下が追加すべきものである。

```
let rec eval1 ctx t = match t with
    ...
  | TmLet(fi,x,v1,t2) when isval ctx v1 →
      termSubstTop v1 t2
  | TmLet(fi,x,t1,t2) →
      let t1' = eval1 ctx t1 in
      TmLet(fi, x, t1', t2)
    ...

let rec typeof ctx t =
  match t with
    ...
  | TmLet(fi,x,t1,t2) →
     let tyT1 = typeof ctx t1 in
     let ctx' = addbinding ctx x (VarBind(tyT1)) in
     (typeof ctx' t2)
```

解答 11.5.2. この定義はあまりうまく機能しない。第一に、評価順を変えてしまうことである。E-LetV 規則および E-Let 規則は値呼びの評価順序を規定しており、`let x=t`$_1$ `in t`$_2$ 中の $\mathtt{t}_1$ は、x に代入して $\mathtt{t}_2$ を処理し始める前に、値に簡約しなくてはならない。第二に、型付け規則 T-Let の妥当性はこの変換で保たれる（これは代入補題（補題 9.3.8）から直接導かれる）が、項が正しく型付け**できないことは保たれない**。例えば、正しく型付けできない項

```
let x = unit(unit) in unit
```

は、正しく型付けされた項 `unit` になってしまう。これは x が本体 `unit` に現れないため、正しく型付けできない部分項 `unit(unit)` が単に消滅してしまうためである。

解答 11.8.2. レコードのパターンに型を追加する一つの案は、図 A-1 に要約したものである。一般化された `let` 構文の型付け規則 T-Let は、別の「パターン型付け」関係を参照する。パターン型付け関係は、（アルゴリズム的に見ると）パターンと入力の型を受け取り、成功した場合には、パターン中の変数に対する適切な束縛からなる文脈を返す。T-Let は本体 $\mathtt{t}_2$ の型検査時に、この文脈を現在の文脈 Γ に加えて型検査を行う。（レコードパターンの異なるフィールドに束縛される変数の集合が互いに素であることを全体にわたって仮定する。）

望むならば、レコードパターン型付け規則を少しだけ改良し、マッチ対象の値が実際に持つフィールドよりも少ないフィールドにしか言及しないパターンを許すこともできる、ということに注意されたい。

→ {} let p （型付き） 図 11-8 の拡張

パターン型付け規則	新しい型付け規則 $\boxed{\Gamma \vdash \mathtt{t} : \mathtt{T}}$
$\vdash \mathtt{x} : \mathtt{T} \Rightarrow \mathtt{x{:}T}$ (P-VAR)	$\dfrac{\Gamma \vdash \mathtt{t}_1 : \mathtt{T}_1 \quad \vdash \mathtt{p} : \mathtt{T}_1 \Rightarrow \Delta \quad \Gamma, \Delta \vdash \mathtt{t}_2 : \mathtt{T}_2}{\Gamma \vdash \mathtt{let\ p{=}t_1\ in\ t_2} : \mathtt{T}_2}$ (T-LET)
$\dfrac{\text{各 } i \text{ に対して } \vdash \mathtt{p}_i : \mathtt{T}_i \Rightarrow \Delta_i}{\vdash \{\mathtt{l}_i{=}\mathtt{p}_i\ ^{i\in 1..n}\} : \{\mathtt{k}_j{:}\mathtt{T}_j\ ^{j\in 1..m}\} \Rightarrow \Delta_1, \ldots, \Delta_n}$ (P-RCD)	

▲図 A-1. 型付きのレコードパターン

$$\frac{\{\mathtt{l}_i\ ^{i\in 1..n}\} \subseteq \{\mathtt{k}_j\ ^{j\in 1..m}\} \qquad \forall\ ^{i\in 1..n}.\ \exists\ ^{j\in 1..m}.\ \mathtt{l}_i = \mathtt{k}_j \text{ かつ } \vdash \mathtt{p}_i : \mathtt{T}_j \Rightarrow \Delta_i}{\vdash \{\mathtt{l}_i{=}\mathtt{p}_i\ ^{i\in 1..n}\} : \{\mathtt{k}_j{:}\mathtt{T}_j\ ^{j\in 1..m}\} \Rightarrow \Delta_1, \ldots, \Delta_n} \quad \text{(P-RCD')}$$

この規則を採用するならば、レコード射影演算子は以下のように糖衣構文として扱えるので、プリミティブな構文としては除いてしまってもよい。

$$\mathtt{t.l} \stackrel{\text{def}}{=} \mathtt{let\ \{l{=}x\}{=}t\ in\ x}$$

拡張した体系に対する型保存定理の証明は、単純型付きラムダ計算の場合とほぼ同じである。唯一追加しなくてはならないのは、パターン型付け関係と実行時のマッチ演算を対応させる補題である。σ が代入であり、Δ が σ と同じ定義域を持つ文脈とする。このとき、$\Gamma \vdash \sigma \models \Delta$ は各 $\mathtt{x} \in dom(\Delta)$ について $\Gamma \vdash \sigma(\mathtt{x}) : \Delta(\mathtt{x})$ であることを意味する。

補題 $\Gamma \vdash \mathtt{t} : \mathtt{T}$ かつ $\vdash \mathtt{p} : \mathtt{T} \Rightarrow \Delta$ ならば、$match(\mathtt{p}, \mathtt{t}) = \sigma$ かつ $\Gamma \vdash \sigma \models \Delta$ □

ここで追加的に導入した記法のために、標準的な代入補題（補題 9.3.8）を少しだけ一般化する必要がある。

補題 $\Gamma, \Delta \vdash \mathtt{t} : \mathtt{T}$ かつ $\Gamma \vdash \sigma \models \Delta$ ならば $\Gamma \vdash \sigma\mathtt{t} : \mathtt{T}$ □

これで、これらの補題を T-LET の場合に用いれば、T-LET 規則の型保存はこれまでと同じパターンの議論で示すことができる[†3]。

解答 11.9.1.

`Bool`	$\stackrel{\text{def}}{=}$	`Unit+Unit`
`true`	$\stackrel{\text{def}}{=}$	`inl unit`
`false`	$\stackrel{\text{def}}{=}$	`inr unit`
$\mathtt{if\ t_0\ then\ t_1\ else\ t_2}$	$\stackrel{\text{def}}{=}$	$\mathtt{case\ t_0\ of\ inl\ x_1 \Rightarrow t_1\ \vert\ inr\ x_2 \Rightarrow t_2}$ ただし $\mathtt{x}_1$ と $\mathtt{x}_2$ はフレッシュ

[†3]（訳注）進行については、図 11-8 の E-LETV における $match(\mathtt{p}, \mathtt{v}_1)$ が、型付けより常に定義されていることがポイントとなる。

解答 11.11.1.

```
equal =
  fix
    (λeq:Nat→Nat→Bool.
       λm:Nat. λn:Nat.
         if iszero m then iszero n
         else if iszero n then false
         else eq (pred m) (pred n));
▸ equal : Nat → Nat → Bool

plus = fix (λp:Nat→Nat→Nat.
              λm:Nat. λn:Nat.
                if iszero m then n else succ (p (pred m) n));
▸ plus : Nat → Nat → Nat

times = fix (λt:Nat→Nat→Nat.
               λm:Nat. λn:Nat.
                 if iszero m then 0 else plus n (t (pred m) n));
▸ times : Nat → Nat → Nat

factorial = fix (λf:Nat→Nat.
                   λm:Nat.
                    if iszero m then 1 else times m (f (pred m)));
▸ factorial : Nat → Nat

factorial 5;
▸ 120 : Nat
```

解答 11.11.2.

```
letrec plus : Nat→Nat→Nat =
  λm:Nat. λn:Nat.
    if iszero m then n else succ (plus (pred m) n) in
letrec times : Nat→Nat→Nat =
  λm:Nat. λn:Nat.
    if iszero m then 0 else plus n (times (pred m) n) in
letrec factorial : Nat→Nat =
  λm:Nat.
    if iszero m then 1 else times m (factorial (pred m)) in
factorial 5;
▸ 120 : Nat
```

解答 11.12.1. 驚くべし。実際には進行定理は**成り立たない**。例えば、式 head[T] nil[T] は行き詰まり状態になる（これにはどの評価規則も適用できないが、かといって値というわけでもない）。本格的なプログラミング言語では、このような状況を行き詰まり状態にするのではなく、例外を発生させることにより処理している。例外については第 14 章で扱う。

解答 11.12.2. すべてを取り除くことはできない。nil への注記を消去すると、型の一意性定理が成り立たなくなる。操作的には、型検査器が nil を見つけると、ある T についての List T という型を割り当てなくてはならないことはわかる。しかし、T として何を選ぶかは、何らかの形の推測を行わずには知ることができない。もちろん、例えば OCaml コンパイラで用いられているような、より洗練された型付けアルゴリズムは、まさにこの種の推測を行っている。この点については第 22 章で再度扱う。

解答 12.1.1. 適用の場合で失敗する。$\mathtt{t_1\ t_2}$ が正規化可能であることの証明に挑戦しているとしよう。帰納法の仮定により $\mathtt{t_1}$ と $\mathtt{t_2}$ は両方とも正規化可能であるとわかる。それぞれの正規形を $\mathtt{v_1}$、$\mathtt{v_2}$ とする。型付け関係の逆転補題（補題 9.3.1）により、$\mathtt{v_1}$ はある型 $\mathtt{T_{11}}$ と $\mathtt{T_{12}}$ について型 $\mathtt{T_{11}{\to}T_{12}}$ を持つ。よって標準形補題（補題 9.3.4）により、$\mathtt{v_1}$ は $\lambda\mathtt{x{:}T_{11}.t_{12}}$ という形を持つ。しかし $\mathtt{t_1\ t_2}$ を $(\lambda\mathtt{x{:}T_{11}.t_{12}})\ \mathtt{v_2}$ に簡約すると、正規形ではなくなる。これは E-AppAbs 規則が適用でき、$[\mathtt{x} \mapsto \mathtt{v_2}]\mathtt{t_{12}}$ となるからである。よって証明を完遂するには、この項が正規化可能であることを示さなければならない。しかしここで行き詰まってしまう。なぜならこの項は、一般に、元の項 $\mathtt{t_1\ t_2}$ より大きいからである（代入によって、$\mathtt{t_{12}}$ 中に $\mathtt{x}$ が出現する数だけ $\mathtt{v_2}$ のコピーができてしまうため）。

解答 12.1.7. 定義 12.1.2 を拡張して節を二つ追加する。

- $R_{\mathtt{Bool}}(\mathtt{t})$ であるのは、$\mathtt{t}$ が停止するとき、かつそのときのみである。
- $R_{\mathtt{T_1\times T_2}}(\mathtt{t})$ であるのは、$\mathtt{t}$ が停止し、$R_{\mathtt{T_1}}(\mathtt{t.1})$ かつ $R_{\mathtt{T_2}}(\mathtt{t.2})$ であるとき、かつそのときのみである。

補題 12.1.4 の証明を拡張して節を一つ追加する。

- ある型 $\mathtt{T_1}$ と $\mathtt{T_2}$ について $\mathtt{T = T_1 \times T_2}$ とする。十分性（$\Longrightarrow$）については、$\mathtt{t} \longrightarrow \mathtt{t'}$ のとき、$R_{\mathtt{T}}(\mathtt{t})$ を仮定して $R_{\mathtt{T}}(\mathtt{t'})$ を証明する必要がある。$R_{\mathtt{T_1\times T_2}}$ の定義から $R_{\mathtt{T_1}}(\mathtt{t.1})$ かつ $R_{\mathtt{T_2}}(\mathtt{t.2})$ であることはわかっている。しかし評価規則 E-Proj1 および E-Proj2 から、$\mathtt{t.1} \longrightarrow \mathtt{t'.1}$ かつ $\mathtt{t.2} \longrightarrow \mathtt{t'.2}$ とわかる。よって帰納法の仮定により、$R_{\mathtt{T_1}}(\mathtt{t'.1})$ かつ $R_{\mathtt{T_2}}(\mathtt{t'.2})$。これらから、$R_{\mathtt{T_1\times T_2}}$ の定義より $R_{\mathtt{T_1\times T_2}}(\mathtt{t'})$ となる。必要性の証明も同様である。

最後に、補題 12.1.5 の証明に（型付け規則ごとに）場合分けを追加する必要がある。

T-If の場合 $\mathtt{t = if\ t_1\ then\ t_2\ else\ t_3}$ $\Gamma \vdash \mathtt{t_1 : Bool}$
$\Gamma \vdash \mathtt{t_2 : T}$ $\Gamma \vdash \mathtt{t_3 : T}$
ただし $\Gamma = \mathtt{x_1{:}T_1, \ldots, x_n{:}T_n}$

$\sigma = [\mathtt{x_1} \mapsto \mathtt{v_1}]\cdots[\mathtt{x_n} \mapsto \mathtt{v_n}]$ とする。帰納法の仮定により、$R_{\mathtt{Bool}}(\sigma\mathtt{t_1})$、$R_{\mathtt{T}}(\sigma\mathtt{t_2})$、$R_{\mathtt{T}}(\sigma\mathtt{t_3})$ を得る。$R_{\mathtt{Bool}}(\sigma\mathtt{t_1})$ から、$\sigma\mathtt{t_1}$ は停止してブール値になることがわかる。よって、$\sigma\mathtt{t}$ は $\sigma\mathtt{t_2}$ か $\sigma\mathtt{t_3}$ に評価される。評価の下での R の保存（補題 12.1.4）より、$R_{\mathtt{T}}(\sigma\mathtt{t_2})$ かつ $R_{\mathtt{T}}(\sigma\mathtt{t_3})$ であることを示せば十分であるが、これらは既に帰納法の仮定によって得られている。

T-True の場合 $\mathtt{t = true}$ $\mathtt{T = Bool}$
直ちに得られる。

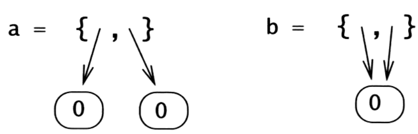
T-False の場合 $\mathtt{t = false}$ $\mathtt{T = Bool}$
直ちに得られる。

T-Pair の場合 $\mathtt{t = \{t_1,t_2\}}$ $\Gamma \vdash \mathtt{t_1 : T_1}$ $\Gamma \vdash \mathtt{t_2 : T_2}$ $\mathtt{T = T_1 \times T_2}$
帰納法の仮定により、$i = 1,2$ について $R_{\mathtt{T}_i}(\sigma\mathtt{t}_i)$。各 $\sigma\mathtt{t}_i$ の正規形を $\mathtt{w}_i$ とする。補題 12.1.4 より、$R_{\mathtt{T}_i}(\mathtt{w}_i)$ であることに注意されたい。
第一射影、第二射影の評価規則により、$\mathtt{\{\sigma t_1,\sigma t_2\}.i} \longrightarrow^* \mathtt{w}_i$ とわかる。ここから補題 12.1.4 により、$R_{\mathtt{T}_i}(\mathtt{\{\sigma t_1,\sigma t_2\}.i})$ を得られる。$R_{\mathtt{T_1\times T_2}}$ の定義から $R_{\mathtt{T_1\times T_2}}(\mathtt{\{\sigma t_1,\sigma t_2\}})$ が得られ、つまり $R_{\mathtt{T_1\times T_2}}(\sigma\mathtt{\{t_1,t_2\}})$ である。

T-Proj1 の場合 $\mathtt{t = t_0.1}$ $\Gamma \vdash \mathtt{t_0 : T_1 \times T_2}$ $\mathtt{T = T_1}$
$R_{\mathtt{T_1\times T_2}}$ の定義から直ちに得られる。

T-PROJ2 の場合　$\mathtt{t} = \mathtt{t}_0\mathtt{.2}$　　$\Gamma \vdash \mathtt{t}_0 : \mathtt{T}_1 \times \mathtt{T}_2$　　$\mathtt{T} = \mathtt{T}_2$
同様。

解答 13.1.1.



解答 13.1.2.　動作は変わる。`update` に与えられたものと異なる添字を渡すと、`lookup` の呼び出しは発散するようになる。重要な点は、`a` に格納された値を取り出す操作を、その値が新しい関数で上書きされる**前に**行う必要があるということである。さもなくば、この取り出しによって得られる値が、元の関数ではなく新しい関数となってしまう。

解答 13.1.3.　言語がプリミティブ `free` を持つと仮定する。`free` は参照セルを引数に取って、その記憶域を解放するものとし、その直後に行われる参照セルの確保は、メモリの同じ部分を得るものとする。このとき次のプログラムは、行き詰まり状態の項 `succ false` へと評価される。

```
let r = ref 0 in
let s = r in
free r;
let t = ref true in
t := false;
succ (!s)
```

この問題設定において、別名付けが重要な役割を担っていることに注意されたい。なぜならば、別名付けのせいで、単に変数 `r` が残りの式で自由に出現したら `free r` を不正とするだけでは、不正な解放を検出できないためである。

このような例は、手動のメモリ管理ができる C のような言語では、容易に再現できる（構築子 `ref` は C の `malloc` に、この例の `free` は C の `free` を単純化したものに、それぞれ対応する）。

解答 13.3.1.　ガベージコレクションを単純に扱うと次のようになるだろう。

(1) 位置の集合 $\mathcal{L}$ が有限であるとして、メモリの有限性をモデル化する[†4][†5]。

(2) ストアにおける位置の**到達可能性**を次のように定義する。`t` で言及されている位置の集合を $locations(\mathtt{t})$ と書く。位置 l と l' およびストア μ について、$l' \in locations(\mu(l))$ であるとき、l' は、μ において l から **1 ステップ到達可能**であるという。l から始まって l' で終わる位置の有限列で、各要素が一つ前の要素から 1 ステップ到達可能であるようなものがあるとき、l' は l から**到達可能**であるという。最後に、ストア μ において項 `t` から到達可能な位置の集合 $reachable(\mathtt{t}, \mu)$ を、μ において $locations(\mathtt{t})$ から到達可能な位置の集合と定義する。

(3) ガベージコレクションの動作を、関係 $\mathtt{t} \mid \mu \longrightarrow_{gc} \mathtt{t} \mid \mu'$ としてモデル化する。この関係は次の規則によって定義される。

$$\frac{\mu' = \mu \text{ の } reachable(\mathtt{t}, \mu) \text{ への制限}}{\mathtt{t} \mid \mu \longrightarrow_{gc} \mathtt{t} \mid \mu'} \qquad \text{(E-GC)}$$

---

†4 （訳注：原著正誤表より）新たな位置 l は $\mathcal{L}$ の要素でなければならないと、規則 E-REFV に書き加える必要があるだろう。

†5 （訳注：原著正誤表より）メモリが有限の場合に限らなければいけないということはない。無限のメモリに対するガベージコレクションについて述べることもできる。しかし、メモリは有限のほうが説明の (5)-(b) はより興味深い内容になる。

（すなわち、μ' の定義域はちょうど $\mathit{reachable}(\mathtt{t},\mu)$ であり、この定義域の任意の位置において、μ' の値は μ の値に等しい。）

(4) 評価の列を、通常の評価ステップの列に、ガベージコレクションがちりばめられたものとして、$\xrightarrow{gc}{}^{*} \stackrel{\text{def}}{=} (\longrightarrow \cup \longrightarrow_{gc})^{*}$ と定義する。規則 E-GC を、1 ステップ評価関係に単純に加えようとしているのでは**ない**ことに注意されたい。つまりガベージコレクションの実行が、評価されようとしている項全体を見ることのできる、「最も外側のレベル」でのみ行われることが重要である。例えば、適用の左側に対する評価の「内部」でガベージコレクションを許してしまうと、評価の右側で実際に言及されている位置を誤って再利用してしまう可能性がある。左側を見ても、項全体からそのような位置がまだ到達可能であることは、わからないためである。

(5)[†6]評価の定義についての以上の変更を正当化するために、これらの変更が最終的な結果に影響しないことを示す。ただし、メモリが使い果たされる可能性は導入される。

(a) $\mathtt{t} \mid \mu \xrightarrow{gc}{}^{*} \mathtt{t}' \mid \mu''$ ならば、ある μ' が存在して、$\mathtt{t} \mid \mu \longrightarrow^{*} \mathtt{t}' \mid \mu'$ となる。ただし μ' は μ'' より大きな定義域を持ち、両方で定義される値については一致する。

(b) $\mathtt{t} \mid \mu \longrightarrow^{*} \mathtt{t}' \mid \mu'$ ならば、次のいずれかが成り立つ。

i. ある μ'' が存在して、$\mathtt{t} \mid \mu \xrightarrow{gc}{}^{*} \mathtt{t}' \mid \mu''$ となる。ただし μ'' は μ' より小さな定義域を持ち、両方で定義される値については一致する。

ii. $\mathtt{t} \mid \mu$ の評価がメモリを使い果たす。すなわち、評価の結果として状態 $\mathtt{t}''' \mid \mu'''$ に至り、$\mathtt{t}'''$ を評価する次のステップが新たな位置の確保を必要とするものの、$\mathit{reachable}(\mathtt{t}''',\mu''') = \mathcal{L}$ となっており、利用可能な位置が残っていない。

ガベージコレクションのこの単純な扱いは、現実のストアの、幾つかの側面を無視している。例えば、異なる種類の値を格納するためには、普通は異なる量のメモリが必要となる。また現実のストアには、**ファイナライザ**（データ構造に付属するコード片で、そのデータ構造が実行時システムによるガベージコレクションを受ける直前に、実行されるもの）や**弱いポインタ**（ポインタではあるが、データ構造への「真の参照」とはみなされないもの。したがってその構造は、それを指す弱いポインタが存在する場合でも、ガベージコレクションの対象となりうる）といった、幾つかのより先進的な機能もある。ガベージコレクションについての、操作的意味論による、より洗練された扱いは、Morrisett et al. [327] に見ることができる。

解答 13.4.1.

```
let r₁ = ref (λx:Nat.0) in
let r₂ = ref (λx:Nat.(!r₁)x) in
(r₁ := (λx:Nat.(!r₂)x);
 r₂);
```

解答 13.5.2. μ を、ただ一つの位置 l を持つストアとする。

$$\mu \quad = \quad (\mathtt{l} \mapsto \lambda\mathtt{x}{:}\mathtt{Unit}.\ (!\mathtt{l})(\mathtt{x}))$$

また、Γ を空の文脈とする。すると μ は、次の二つのストア型付けのどちらに関しても、正しく型

[†6] （訳注：原著正誤表より）この (5) の文章はややわかりにくい。(a) で言いたいのは、ガベージコレクションありでの評価が可能ならば、同じ評価が「無限のストアの下で」、ガベージコレクションなしで可能だということである。(b) についても同様である。

（同じプログラムを「有限の」ストアで、ガベージコレクションありとなしの条件で実行した場合も、比較すると面白いだろう。例えば、ガベージコレクションなしでプログラムの実行が完了した場合、それはガベージコレクションありの場合と同じ結果に至り、違いは位置の名前の付け替えだけのはずである。）

また、進行定理と保存定理はストア型付けに依存していたが、現在の保存定理の主張においては、ストア型付けは増大するだけであることに注意されたい。ガベージコレクションを含めると、「ストア型付けにガベージコレクションを行う」こともできなければならない。

付けされる。

$$\Sigma_1 \quad = \quad \texttt{l:Unit→Unit}$$
$$\Sigma_2 \quad = \quad \texttt{l:Unit→(Unit→Unit)}$$

解答 13.5.8. 正しく型付けされていて正規化可能性を持たない項が、この体系には存在する。演習 13.1.2 に一つの例が出てきたが、もう一つ挙げよう。

```
t1 = λr:Ref (Unit→Unit).
        (r := (λx:Unit. (!r)x);
         (!r) unit);
t2 = ref (λx:Unit. x);
```

`t1` を `t2` に適用すると、（正しく型付けされた）発散する項ができる。

より一般に、次の手順を追うことで、任意の再帰関数を参照を用いて定義できる。（この技法は、幾つかの関数型言語の実装で実際に用いられている。）

(1) `ref` セルを確保し、適切な型を持つダミーの関数で初期化する。

```
fact_ref = ref (λn:Nat.0);
▸ fact_ref : Ref (Nat→Nat)
```

(2) 求めている関数の本体を定義する。再帰呼び出しを作るために、参照セルの中身を使う。

```
fact_body =
   λn:Nat.
      if iszero n then 1 else times n ((!fact_ref)(pred n));
▸ fact_body : Nat → Nat
```

(3) 参照セルへ実際の関数本体を格納し、参照を張り直す。

```
fact_ref := fact_body;
```

(4) 参照セルの中身を取り出すと、望む通りに使うことができる。

```
fact = !fact_ref;
fact 5;
▸ 120 : Nat
```

解答 14.1.1. `error` に意図する型を注釈すると、型保存の性質が壊れてしまう。例えば以下の項

```
(λx:Nat.x) ((λy:Bool.5) (error as Bool));
```

は正しく型付けされている（ただし `error as T` は例外の型注釈の構文）。しかしこれを 1 ステップ評価すると、正しく型付けされない項になる。

```
(λx:Nat.x) (error as Bool);
```

評価規則は `error` を、発生箇所からプログラムのトップレベルまで伝播するので、`error` はさまざまな型を持つと考えられる。規則 T-ERROR の柔軟性がこれを可能にしている。

解答 14.3.1. Standard ML の定義（Milner, Tofte, Harper, and MacQueen [316]; Milner and Tofte [314]）が `exn` 型を形式化している。関連する扱いが Harper and Stone [199] に見られる。

解答 14.3.2. Leroy and Pessaux [282] を参照せよ。

解答 14.3.3. Harper et al. [194] を参照せよ。

解答 15.2.1.

$$\dfrac{\dfrac{}{\begin{array}{l}\texttt{\{x:Nat,y:Nat,z:Nat\}}\\ \texttt{<: \{y:Nat,x:Nat,z:Nat\}}\end{array}}\textsc{S-RcdPerm} \qquad \dfrac{}{\begin{array}{l}\texttt{\{y:Nat,x:Nat,z:Nat\}}\\ \texttt{<: \{y:Nat\}}\end{array}}\textsc{S-RcdWidth}}{\texttt{\{x:Nat,y:Nat,z:Nat\} <: \{y:Nat\}}}\textsc{S-Trans}$$

解答 15.2.2. 同じ結論を持つ導出は他にもたくさんある。例えば以下。

$$\dfrac{\dfrac{\dfrac{\vdots}{\vdash \texttt{f : Rx}\to\texttt{Nat}} \qquad \dfrac{\dfrac{}{\texttt{Rxy <: Rx}}\textsc{S-RcdWidth} \quad \dfrac{}{\texttt{Nat <: Nat}}\textsc{S-Refl}}{\texttt{Rx}\to\texttt{Nat <: Rxy}\to\texttt{Nat}}\textsc{S-Arrow}}{\vdash \texttt{f : Rxy}\to\texttt{Nat}}\textsc{T-Sub} \qquad \dfrac{\vdots}{\vdash \texttt{xy : Rxy}}}{\vdash \texttt{f xy : Nat}}\textsc{T-App}$$

以下は他の例。

$$\dfrac{\dfrac{\vdots}{\vdash \texttt{f : Rx}\to\texttt{Nat}} \qquad \dfrac{\dfrac{\vdots}{\vdash \texttt{xy : Rxy}}\textsc{T-Rcd} \qquad \dfrac{\dfrac{}{\texttt{Rxy <: Rx}}\textsc{S-RcdWidth} \quad \dfrac{}{\texttt{Rx <: Rx}}\textsc{S-Refl}}{\texttt{Rxy <: Rx}}\textsc{S-Trans}}{\vdash \texttt{xy : Rx}}\textsc{T-Sub}}{\vdash \texttt{f xy : Nat}}\textsc{T-App}$$

実際、二つ目の例が示唆するように、この計算体系で導出可能な**任意の**型付け判断式には、**無限に多くの**導出が存在する。

解答 15.2.3.

(1) 六つ存在する。{a:Top,b:Top}、{b:Top,a:Top}、{a:Top}、{b:Top}、{}、Top。

(2) 例えば以下。

S_0 = {}
S_1 = {a:Top}
S_2 = {a:Top,b:Top}
S_2 = {a:Top,b:Top,c:Top}
以下同様。

(3) 例えば以下。$T_0 = S_0 \to$Top、$T_1 = S_1 \to$Top、$T_2 = S_2 \to$Top、以下同様。

解答 15.2.4. (1) 存在しない。仮にそのような型が存在するとすれば、矢印型とレコード型のいずれかでなければならないだろう（明らかに Top ではない）。しかしレコード型はどの矢印型の部分型でもないし、その反対も同様である。(2) やはり存在しない。仮にそのような矢印型 $T_1 \to T_2$ が存在するとすれば、その定義域の型 T_1 は他のすべての型 S_1 の部分型でなければならない。しかしこれはありえないと言ったばかりである。

解答 15.2.5. 既存の評価意味論を変えたくないなら、この規則を追加するのはやめたほうがよいだろう。この新しい規則があると、例えば Nat×Bool <: Nat が導出可能になる。つまり行き詰まり状態の項 (succ (5,true)) が正しく型付けされてしまうので、進行定理に違反する。「型強制意味論」（15.6 節を見よ）の下ではこの規則は安全である。ただしアルゴリズム的な部分型検査で少し問題が起きる。

解答 15.3.1. 部分型関係に誤った二つ組を追加すると、保存も進行も壊れてしまう。例えば、以下の公理を追加するとする。

$$\{x:\{\}\} <: \{x:\mathsf{Top}\to\mathsf{Top}\}$$

すると、$t = (\{x=\{\}\}.x)\{\}$ として $\Gamma \vdash t : \mathsf{Top}$ が導出できる。しかし $t \longrightarrow \{\}\{\}$ であり、これは型付けできない。つまり保存に違反してしまった。他の例として、以下の公理を追加するとする。

$$\{\} <: \mathsf{Top}\to\mathsf{Top}$$

すると項 {}{} が**型付け可能**になるが、この項は行き詰まり状態だが値ではない。つまり、進行に違反してしまった。

一方、部分型関係から二つ組を取り除くことは、実際のところ問題ない。進行の性質の言明において型付け関係が言及される箇所は前提の中しかないので、部分型関係（ひいては型付け関係）を制限しても進行の成立は簡単になるだけである。保存の性質については、特に推移律を取り除くと問題が起きるのではないかと心配になるかもしれない。問題が起きない直観的な理由は、この型システムにおける推移律の役割が、包摂規則 T-SUB を用いて推移律を復元できるという意味で実際には本質的でないためである。例えば、

$$\dfrac{\dfrac{\vdots}{\Gamma \vdash t : S} \qquad \dfrac{\dfrac{\vdots}{S <: U} \quad \dfrac{\vdots}{U <: T}}{S <: T}\ \text{S-TRANS}}{\Gamma \vdash t : T}\ \text{T-SUB}$$

という導出は、常に以下のように書き直せる。

$$\dfrac{\dfrac{\dfrac{\vdots}{\Gamma \vdash t : S} \quad \dfrac{\vdots}{S <: U}}{\Gamma \vdash t : U}\ \text{T-SUB} \qquad \dfrac{\vdots}{U <: T}}{\Gamma \vdash t : T}\ \text{T-SUB}$$

解答 15.3.2.

(1) 部分型付け導出に関する帰納法による。図 15-1 と図 15-3 の部分型付け規則をよく見ると、明らかに、$S <: T_1\to T_2$ の導出の最後の規則は S-REFL、S-TRANS、S-ARROW のいずれかでなければならない。
最後の規則が S-REFL であれば、結論は直ちに得られる。（この場合は $S = T_1\to T_2$ で、反射律により $T_1 <: T_1$ も $T_2 <: T_2$ も導出できるためである。）
最後の規則が S-TRANS であれば、ある型 U が存在して $S <: U$ を結論に持つ部分導出と $U <: T_1\to T_2$ を結論に持つ部分導出がある。二つ目の部分導出に帰納法の仮定を適用して、U が $U_1\to U_2$ という形を持ち、$T_1 <: U_1$ かつ $U_2 <: T_2$ であることがわかる。ここで、U が矢印型であることを知っているので、帰納法の仮定を一つ目の部分導出に適用して、$S = S_1\to S_2$ を得られる（ただし $U_1 <: S_1$ かつ $S_2 <: U_2$）。最後に S-TRANS を二回用いて、証明中の事実を再構築できる。つまり $T_1 <: U_1$ と $U_1 <: S_1$ から $T_1 <: S_1$ を得て、さらに $S_2 <: U_2$ と $U_2 <: T_2$ から $S_2 <: T_2$ を得ればよい。
最後の規則が S-ARROW であれば、S は所望の形を持つ。そして直接の部分導出が、S_1 と S_2 について必要な事実そのものである。

(2) 部分導出に関する帰納法による。ここでも、部分型付け規則をよく見れば、$S <: \{l_i:T_i\ ^{i\in 1..n}\}$ の導出の最後の規則は S-REFL、S-TRANS、S-RCDWIDTH、S-RCDDEPTH、S-RCDPERM のいずれ

かであるとわかる。S-REFL の場合は自明である。S-RCDWIDTH、S-RCDDEPTH、S-RCDPERM の場合はどれも直ちに証明できる。

最後の規則が S-TRANS であれば、ある型 U が存在して、S <: U を結論に持つ部分導出と、U <: $\{l_i : T_i\ ^{i\in 1..n}\}$ を結論に持つ部分導出が存在する。二つ目の部分導出に帰納法の仮定を適用すると、U が $\{u_a : U_a\ ^{a\in 1..o}\}$ という形を持つとわかる（ただし $\{l_i\ ^{i\in 1..n}\} \subseteq \{u_a\ ^{a\in 1..o}\}$、かつすべての $l_i = u_a$ に対して U_a <: T_i)。ここで、U はレコード型を持つと知っているから、帰納法の仮定を一つ目の部分導出に適用して、$S = \{k_j : S_j\ ^{j\in 1..m}\}$ が得られる（ただし $\{u_a\ ^{a\in 1..o}\} \subseteq \{k_j\ ^{j\in 1..m}\}$、かつすべての $u_a = k_j$ に対して S_j <: U_a)。これらの事実を組み合わせて、集合の包含の推移律により $\{l_i\ ^{i\in 1..n}\} \subseteq \{k_j\ ^{j\in 1..m}\}$ を得られる。また S-TRANS より、すべての $l_i = k_j$ に対して S_j <: T_i である。なぜなら、T のすべてのラベルは U のラベルでもある（つまり、l_i が u_a と同じであるような a が存在する）ので、S_j <: U_a かつ U_a <: T_i とわかる。（ここでメタ変数の名前の選び方が変なのはやむをえなかった。アルファベットを順番に選ぶには文字が足りなかっただけである。）

解答 15.3.6. どちらも型付け導出に関する帰納法によって進む。一つ目のほうだけ議論する。

型付け規則をよく見ると、明らかに、⊢ v : $T_1 \to T_2$ の導出の最後の規則は T-ABS か T-SUB のいずれかでなければならない。T-ABS だとしたら、所望の結論はこの規則の前提から直ちに得られる。よって最後の規則が T-SUB だとしよう。

T-SUB の前提から、⊢ v : S かつ S <: $T_1 \to T_2$ である。逆転の補題（補題 15.3.2）から、S は $S_1 \to S_2$ という形を持つとわかる。すると結論は帰納法の仮定により得られる。

解答 16.1.2. (1) は S の構造に関する単純な帰納法。

(2) のためには、まず、(1) より、S <: T の任意の導出が存在するならば反射律を用いない導出が存在する、ということに注目する。これから、S <: T の反射律を用いない導出のサイズに関する帰納法を行う。今までのような導出の構造に関する帰納法ではなく、導出の**サイズ**に関する帰納法であることに注意されたい。こうする必要があるのは、矢印のケースとレコードのケースにおいて、元の導出の部分導出として現れるものではない、**新しく構築した**導出に対して帰納法の仮定を適用するためである。

導出の最後の規則が S-TRANS 以外であれば、結論は帰納法の仮定により直ちに導かれる（つまり、帰納法の仮定により、最後の規則の部分導出をすべて推移律を含まない導出に置き換えることができる。最後の規則自体も推移律でないから、導出全体が推移律を含まないものとなる)。よって最後の規則が S-TRANS であるとする。つまり、ある型 U について、結論に S <: U と U <: T を持つ部分導出が二つ与えられる。これらの部分導出両方の最後の規則の二つ組を場合分けして進める。

任意の規則/S-TOP の場合　T = Top

右側の導出が S-TOP で終わるならば、結論は直ちに得られる。なぜならば、S が何であろうと S <: Top は S-TOP によって導出できるためである。

S-TOP/任意の規則の場合　U = Top

左側の導出が S-TOP で終わるならば、まず注目するのは、帰納法の仮定により右側の導出が推移律を含まないと仮定できることである。ここで、部分型付け規則をよく調べると、この部分導出の最後の規則は S-TOP でなければならないとわかる（反射律は既に除去しており、他の規則はどれも U が矢印かレコードであることを要求している)。結論は S-TOP により得られる。

S-ARROW/S-ARROW の場合　$S = S_1 \to S_2$　$U = U_1 \to U_2$　$T = T_1 \to T_2$
$U_1 <: S_1$　$S_2 <: U_2$
$T_1 <: U_1$　$U_2 <: T_2$

S-TRANS を用いて、与えられた部分導出から $T_1 <: S_1$ と $S_2 <: T_2$ の導出を構築できる。さらに、この新しい導出はどちらも元の導出より真に小さい。よって帰納法の仮定を適用して、$T_1 <: S_1$ と $S_2 <: T_2$ の導出で推移律を含まないものを得られる。これらと S-ARROW を組み合わせて、$S_1 \to S_2 <: T_1 \to T_2$ の導出で推移律を含まないものを得られる。

S-RCD/S-RCD の場合
同様。

その他の場合
他の組み合わせ（S-ARROW/S-RCD と S-RCD/S-ARROW）は起きえない。なぜならこれらは U の形に対して両立しない制約を課すためである。

解答 16.1.3. Bool を追加するなら、補題 16.1.2 の最初の部分を少し修正する必要がある。「$S <: S$ は任意の型 S に対して S-REFL を用いずに導出できる。**ただし型 Bool を除く**」とする。他の方法としては、部分型関係の定義に

$$\text{Bool} <: \text{Bool}$$

という規則を追加することで、S-REFL を削除することもできる。

解答 16.2.5. 宣言的な型付け導出に関する帰納法による。導出の最後の規則について場合分けを行う。

T-VAR の場合　$t = x$　$\Gamma(x) = T$
TA-VAR から直ちに得られる。

T-ABS の場合　$t = \lambda x{:}T_1.t_2$　$\Gamma, x{:}T_1 \vdash t_2 : T_2$　$T = T_1 \to T_2$
帰納法の仮定により、ある $S_2 <: T_2$ に対して $\Gamma, x{:}T_1 \mapsto t_2 : S_2$。TA-ABS より、$\Gamma \mapsto t : T_1 \to S_2$。S-ARROW より、所要の通り $T_1 \to S_2 <: T_1 \to T_2$。

T-APP の場合　$t = t_1\ t_2$　$\Gamma \vdash t_1 : T_{11} \to T_{12}$　$\Gamma \vdash t_2 : T_{11}$　$T = T_{12}$
帰納法の仮定により、ある $S_1 <: T_{11} \to T_{12}$ に対して $\Gamma \mapsto t_1 : S_1$ であり、かつ、ある $S_2 <: T_{11}$ に対して $\Gamma \mapsto t_2 : S_2$ である。部分型関係の逆転補題（補題 15.3.2）より、S_1 は、$T_{11} <: S_{11}$ かつ $S_{12} <: T_{12}$ なる S_{11} と S_{12} に対して $S_{11} \to S_{12}$ という形を持つ。推移律により、$S_2 <: S_{11}$。アルゴリズム的部分型付けの完全性により、$\mapsto S_2 <: S_{11}$。ここで、TA-APP より、$\Gamma \mapsto t_1\ t_2 : S_{12}$。これでこの場合は完了である（なぜなら $S_{12} <: T_{12}$ は既に得ているため）。

T-RCD の場合　$t = \{l_i{=}t_i\ ^{i\in 1..n}\}$　各 i に対して $\Gamma \vdash t_i : T_i$
$T = \{l_i{:}T_i\ ^{i\in 1..n}\}$
単純。

T-PROJ の場合　$t = t_1.l_j$　$\Gamma \vdash t_1 : \{l_i{:}T_i\ ^{i\in 1..n}\}$　$T = T_j$
適用の場合と同様。

T-SUB の場合　$t : S$　$S <: T$
帰納法の仮定と部分型関係の推移律による。

解答 16.2.6. 項 $\lambda x{:}\{a{:}Nat\}.x$ は、宣言的な規則の下では型 $\{a{:}Nat\} \to \{a{:}Nat\}$ と $\{a{:}Nat\} \to Top$ のどちらも持つ。しかし S-ARROW がなければ、これらの型は互いの部分型にはならない（これらの型に共通の部分型も存在しない）。

解答 16.3.2. まず、相互に再帰する二つのアルゴリズムを与える。我々の主張（後で証明する）は、これらのアルゴリズムが二つの型 S と T の結び J と交わり M をそれぞれ求めるということである。二つ目のアルゴリズムは失敗することもある。これは S と T が交わりを持たないことを表す。

$$
\mathtt{S} \vee \mathtt{T} \;=\; \begin{cases}
\mathtt{Bool} & \mathtt{S} = \mathtt{T} = \mathtt{Bool}\ \text{の場合} \\
\mathtt{M}_1{\rightarrow}\mathtt{J}_2 & \mathtt{S} = \mathtt{S}_1{\rightarrow}\mathtt{S}_2 \quad \mathtt{T} = \mathtt{T}_1{\rightarrow}\mathtt{T}_2 \\
 & \mathtt{S}_1 \wedge \mathtt{T}_1 = \mathtt{M}_1 \quad \mathtt{S}_2 \vee \mathtt{T}_2 = \mathtt{J}_2 \quad \text{の場合} \\
\{\mathtt{j}_l{:}\mathtt{J}_l\ ^{l\in 1..q}\} & \mathtt{S} = \{\mathtt{k}_j{:}\mathtt{S}_j\ ^{j\in 1..m}\} \\
 & \mathtt{T} = \{\mathtt{l}_i{:}\mathtt{T}_i\ ^{i\in 1..n}\} \\
 & \{\mathtt{j}_l\ ^{l\in 1..q}\} = \{\mathtt{k}_j\ ^{j\in 1..m}\} \cap \{\mathtt{l}_i\ ^{i\in 1..n}\} \\
 & \text{各}\ \mathtt{j}_l = \mathtt{k}_j = \mathtt{l}_i\ \text{に対して}\ \mathtt{S}_j \vee \mathtt{T}_i = \mathtt{J}_l \quad \text{の場合} \\
\mathtt{Top} & \text{それ以外の場合}
\end{cases}
$$

$$
\mathtt{S} \wedge \mathtt{T} \;=\; \begin{cases}
\mathtt{S} & \mathtt{T} = \mathtt{Top}\ \text{の場合} \\
\mathtt{T} & \mathtt{S} = \mathtt{Top}\ \text{の場合} \\
\mathtt{Bool} & \mathtt{S} = \mathtt{T} = \mathtt{Bool}\ \text{の場合} \\
\mathtt{J}_1{\rightarrow}\mathtt{M}_2 & \mathtt{S} = \mathtt{S}_1{\rightarrow}\mathtt{S}_2 \quad \mathtt{T} = \mathtt{T}_1{\rightarrow}\mathtt{T}_2 \\
 & \mathtt{S}_1 \vee \mathtt{T}_1 = \mathtt{J}_1 \quad \mathtt{S}_2 \wedge \mathtt{T}_2 = \mathtt{M}_2 \quad \text{の場合} \\
\{\mathtt{m}_l{:}\mathtt{M}_l\ ^{l\in 1..q}\} & \mathtt{S} = \{\mathtt{k}_j{:}\mathtt{S}_j\ ^{j\in 1..m}\} \\
 & \mathtt{T} = \{\mathtt{l}_i{:}\mathtt{T}_i\ ^{i\in 1..n}\} \\
 & \{\mathtt{m}_l\ ^{l\in 1..q}\} = \{\mathtt{k}_j\ ^{j\in 1..m}\} \cup \{\mathtt{l}_i\ ^{i\in 1..n}\} \\
 & \text{各}\ \mathtt{m}_l = \mathtt{k}_j = \mathtt{l}_i\ \text{に対して}\ \mathtt{S}_j \wedge \mathtt{T}_i = \mathtt{M}_l \\
 & \mathtt{m}_l = \mathtt{k}_j\ \text{が}\ \mathtt{S}\ \text{のみに現れるならば}\ \mathtt{M}_l = \mathtt{S}_j \\
 & \mathtt{m}_l = \mathtt{l}_i\ \text{が}\ \mathtt{T}\ \text{のみに現れるならば}\ \mathtt{M}_l = \mathtt{T}_i \quad \text{の場合} \\
\mathit{fail} & \text{それ以外の場合}
\end{cases}
$$

一つ目のアルゴリズムの矢印のケースでは、二つ目のアルゴリズムの呼び出しが失敗する可能性がある。この場合は一つ目のアルゴリズムは最後のケースに進み、結果は Top となる。例えば $\mathtt{Bool}{\rightarrow}\mathtt{Top} \vee \{\}{\rightarrow}\mathtt{Top} = \mathtt{Top}$ である。

$\vee$ と $\wedge$ が全域関数である（つまり発散しない）ということは容易に確認できる。S と T の合計サイズが再帰呼び出しのたびに常に減少するということに注目すればよい。さらに $\wedge$ のアルゴリズムは、入力が下界を持つならば決して失敗しない。

補題 A.10. L <: S かつ L <: T ならば、ある M が存在して $\mathtt{S} \wedge \mathtt{T} = \mathtt{M}$ □

［証明］ S のサイズに関する帰納法による（T でもよい）。S と T の形に基づく場合分けを行う。S または T が Top ならば、$\wedge$ の定義の先頭二つのどちらかのケースが適用でき、その結果はそれぞれ T か S である。S と T が異なる形を持つケースは起きえない。なぜなら部分型関係の逆転補題（補題 15.3.2）が L の形に矛盾した要求をすることになるからである。例えば S が矢印の場合、L も矢印でなければならないが、一方で T がレコードなら、L はレコードでもなければならない[†7]。

S と T が共に Bool ならば、定義の三つ目のケースが適用でき、証明は終了する。

$\mathtt{S} = \mathtt{S}_1{\rightarrow}\mathtt{S}_2$ かつ $\mathtt{T} = \mathtt{T}_1{\rightarrow}\mathtt{T}_2$ としよう。$\vee$ のアルゴリズムの全域性から、最初の再帰呼び出しは何かしらの型 $\mathtt{J}_1$ を返すことがわかる。一方で逆転補題により、L は $\mathtt{L}_1{\rightarrow}\mathtt{L}_2$ という形を持ち、$\mathtt{L}_2$ <: $\mathtt{S}_2$

[†7] 厳密にいえば、補題 15.3.2 では Bool を扱っていなかった。Bool のケースを追加しても、Bool の唯一の部分型は Bool 自身だというだけである。

かつ L_2 <: T_2 でなければならない。つまり、L_2 は S_2 と T_2 の共通の下界であり、帰納法の仮定が適用でき、その結果 $S_2 \wedge L_2$ は失敗せず、型 M_2 を返すことがわかる。よって $S \wedge T = J_1 \to M_2$ 。

最後に、$S = \{k_j : S_j{}^{j \in 1..m}\}$ かつ $T = \{l_i : T_i{}^{i \in 1..n}\}$ とする。逆転補題により、L は S と T のいずれかが持つラベルをすべて含むレコード型でなければならない。さらに、S と T の**両方**が持つ各ラベルについて、逆転補題により、L の対応するフィールドは S と T のフィールドの共通の部分型であるとわかる。このことから、共通のラベルに対する ∧ のアルゴリズムの再帰呼び出しはすべて成功することが保証される。 □

さて、これらの定義が結びと交わりを計算することを検証しよう。議論は二つに分かれる。命題 A.11 は交わりとして計算された結果が S と T の下界であること、さらに結びが上界であることを示す。命題 A.12 は交わりとして計算された結果が S と T の任意の共通下界より大きく、結びは任意の共通上界より小さいことを示す。

命題 A.11.

(1) もし $S \vee T = J$ ならば、S <: J かつ T <: J
(2) もし $S \wedge T = M$ ならば、M <: S かつ M <: T □

[証明]　$S \wedge T = M$ または $S \vee T = J$ の「導出」のサイズ（つまり M や J を計算するのに ∧ と ∨ の定義を再帰呼び出しする回数）に関する帰納法による。 □

命題 A.12.

(1) $S \vee T = J$ かつ、ある U について S <: U かつ T <: U とする。このとき J <: U
(2) $S \wedge T = M$ かつ、ある L について L <: S かつ L <: T とする。このとき L <: M □

[証明]　(1) と (2) を併せて証明する。S と T のサイズに関する帰納法による（実際、ほぼ何に関する帰納法でもうまくいく）。与えられた S と T に対し、(1) と (2) を順次検討する。

(1) については、U の形に関する場合分けを行う。U が Top ならば、J が何かにかかわらず J <: Top なので完了である。U = Bool ならば、逆転補題（補題 15.3.2）から S と T は共に Bool でなければならないとわかる。よって J = Bool で完了である。他の場合はもっと興味深い。

- $U = U_1 \to U_2$ ならば、逆転補題により、$S = S_1 \to S_2$ かつ $T = T_1 \to T_2$、ただし U_1 <: S_1、U_1 <: T_1、S_2 <: U_2、T_2 <: U_2 である。帰納法の仮定により、S_1 と T_1 の交わり M_1 は U_1 の上に存在する。一方 S_2 と T_2 の結び J_2 は U_2 の下に存在する。S-Arrow より、$M_1 \to J_2$ <: $U_1 \to U_2$。
- U がレコード型ならば、逆転補題により、S と T もレコード型である。さらに S と T のラベルは U のラベルの上位集合であり、U に含まれる各フィールドの型は S と T の対応するフィールドの上位型である。S と T の結びは少なくとも U のラベルを持ち、かつ（帰納法の仮定により）結びのフィールドは U の対応するフィールドの部分型である。

(2) について、再度 S と T の形に関する場合分けを行う。片方が Top の場合、交わりはもう片方であり、結論は直ちに得られる。補題 A.10 の証明で説明したように、S と T が異なる（Top 以外の）形を持つ場合は起きえない。共に Bool の場合も結論は直ちに得られる。残りの場合（S と T が共に矢印、または共にレコード）はもっと興味深い。

- $S = S_1 \to S_2$ かつ $T = T_1 \to T_2$ の場合、逆転補題により、$L = L_1 \to L_2$、ただし S_1 <: L_1、T_1 <: L_1、L_2 <: S_2、L_2 <: T_2 でなければならない。帰納法の仮定により、S_1 と T_1 の結び J_1 は L_1 の下に存在し、S_2 と T_2 の交わり M_2 は L_2 の上に存在する。S-Arrow により、$L_1 \to L_2$ <: $J_1 \to M_2$。

- SとTがレコード型の場合、逆転補題により、Lもレコード型である。さらにLは、SとTが持つラベルをすべて（もしかしたらそれ以上に）持たなければならず、対応するフィールド同士は部分型関係になければならない。ここで、SとTの交わりMの各ラベルm_lについて、三つの可能性がある。m_lがSとTの両方に含まれる場合は、Mにおける型はSとTにおける型の交わりであり、Lにおいて対応する型は帰納法の仮定によりMにおける型の部分型である。一方、m_lがSのみに含まれる場合、Mにおける対応する型はSの型と同じであり、Lにおける型がより小さいことは既にわかっている。m_lがTのみに含まれる場合も同様である。

□

解答 16.3.3. この項の最小型はBoolと{}の結び、つまりTopである。しかし、この項が型付け可能であるということは、おそらく我々の言語の欠点と見るべきだろう。プログラマがこの式を本当に意図して書いたとは想像しがたい。どのみち型Topの値に関して可能な演算はなく、そもそもこれを計算することにはあまり意味がない。この欠点に対する解決は二つ考えられる。一つは単純に、型システムからTopを取り除いて、∨を部分関数にすることである。もう一つは、Topを保持しつつも、最小型がTopになる項が現れたら型検査器が警告を出すようにすることである。

解答 16.3.4. Ref型を扱うのは単純である。単に結びと交わりのアルゴリズムに節を一つ追加すればよい。

$$
\mathsf{S} \vee \mathsf{T} \;=\; \begin{cases} \ldots & \ldots \\ \mathsf{Ref(T_1)} & \mathsf{S} = \mathsf{Ref(S_1)},\ \mathsf{T} = \mathsf{Ref(T_1)},\ \mathsf{S_1} <: \mathsf{T_1},\ \mathsf{T_1} <: \mathsf{S_1}\ \text{の場合} \\ \ldots & \ldots \end{cases}
$$

$$
\mathsf{S} \wedge \mathsf{T} \;=\; \begin{cases} \ldots & \ldots \\ \mathsf{Ref(T_1)} & \mathsf{S} = \mathsf{Ref(S_1)},\ \mathsf{T} = \mathsf{Ref(T_1)},\ \mathsf{S_1} <: \mathsf{T_1},\ \mathsf{T_1} <: \mathsf{S_1}\ \text{の場合} \\ \ldots & \ldots \end{cases}
$$

しかしRefをSourceとSink構築子に詳細化すると、本質的な問題が生じる。部分型関係がもはや結び（と交わりも）を持たなくなるのである。例えば型Ref{a:Nat,b:Bool}とRef{a:Nat}はSource{a:Nat}とSink{a:Nat,b:Bool}両方の部分型だが、これらの型は共通の下界を持たない。

この問題を解決するにはさまざまな方法がある。おそらく最も単純なのは、SourceかSinkの両方ではなく**どちらか一方だけ**を型システムに追加することである。多くのアプリケーションの領域では、これで十分である。例えば、18.12節で説明する詳細化されたクラスの実装ではSourceだけが必要である。一方チャネル型（15.5節）を持つ並行言語では、Sinkだけが欲しいかもしれない。サーバプロセスを定義し、そのアクセスチャネルに「送信する能力」のみを受け渡せるようになるからである（受信する能力はサーバプロセス自身だけが持っていればよい）。

Source型だけなら、結びのアルゴリズムは以下のように詳細化すれば完全なままとできる。（上で説明したRefの節も必要である。交わりのアルゴリズムには同様の節を追加する。）

$$
\mathsf{S} \vee \mathsf{T} = \begin{cases} \ldots & \ldots \\ \mathsf{Source(J_1)}^{\dagger 8} & \mathsf{S} = \mathsf{Ref(S_1)},\ \mathsf{T} = \mathsf{Ref(T_1)},\ \mathsf{S_1} \vee \mathsf{T_1} = \mathsf{J_1}\ \text{の場合} \\ \mathsf{Source(J_1)} & \mathsf{S} = \mathsf{Source(S_1)},\ \mathsf{T} = \mathsf{Source(T_1)},\ \mathsf{S_1} \vee \mathsf{T_1} = \mathsf{J_1}\ \text{の場合} \\ \mathsf{Source(J_1)} & \mathsf{S} = \mathsf{Ref(S_1)},\ \mathsf{T} = \mathsf{Source(T_1)},\ \mathsf{S_1} \vee \mathsf{T_1} = \mathsf{J_1}\ \text{の場合} \\ \mathsf{Source(J_1)} & \mathsf{S} = \mathsf{Source(S_1)},\ \mathsf{T} = \mathsf{Ref(T_1)},\ \mathsf{S_1} \vee \mathsf{T_1} = \mathsf{J_1}\ \text{の場合} \\ \ldots & \ldots \end{cases}
$$

†8（訳注）ただし$\mathsf{Ref(T_1)}$の場合が優先される。

他の解決（Hennessy and Riely [206] が提案した）は、Ref 型構築子を詳細化して、1 引数でなく 2 引数にすることだ。Ref S T の要素は、型 S の要素を**格納**し、型 T の要素を**読み出す**ために用いられる参照セルである。新しい Ref は第一引数に関しては反変であり、第二引数に関しては共変である。このようにすると、Sink S は Ref S Top の略記として、Source T は Ref Bot T の略記として定義できる。

解答 16.4.1.　追加する必要がある。

$$\frac{\Gamma \mapsto t_1 : T_1 \qquad T_1 = \mathtt{Bot} \qquad \Gamma \mapsto t_2 : T_2 \qquad \Gamma \mapsto t_3 : T_3 \qquad T_2 \vee T_3 = T}{\Gamma \mapsto \mathtt{if}\ t_1\ \mathtt{then}\ t_2\ \mathtt{else}\ t_3 : T} \qquad \text{(TA-IF)}$$

以下の規則

$$\frac{\Gamma \mapsto t_1 : T_1 \qquad T_1 = \mathtt{Bot} \qquad \Gamma \mapsto t_2 : T_2 \qquad \Gamma \mapsto t_3 : T_3}{\Gamma \mapsto \mathtt{if}\ t_1\ \mathtt{then}\ t_2\ \mathtt{else}\ t_3 : \mathtt{Bot}} \qquad \text{(TA-IF)}$$

もよさそうに見える。この規則は安全である（Bot は空なので、t_1 の評価は通常の結果を返すことはない）が、宣言的な型付け規則では型付けできない項に型を割り当ててしまう。よってこちらを選ぶと定理 16.2.4 が成り立たなくなる。

解答 17.3.1.　命題 16.3.2 の演習から、アルゴリズムをただ書き起こせばよい。

```
let rec join tyS tyT =
  match (tyS,tyT) with
    (TyArr(tyS1,tyS2),TyArr(tyT1,tyT2)) →
      (try TyArr(meet tyS1 tyT1, join tyS2 tyT2)
        with Not_found → TyTop)
  | (TyBool,TyBool) →
      TyBool
  | (TyRecord(fS), TyRecord(fT)) →
      let labelsS = List.map (fun (li,_) → li) fS in
      let labelsT = List.map (fun (li,_) → li) fT in
      let commonLabels =
        List.find_all (fun l → List.mem l labelsT) labelsS in
      let commonFields =
        List.map (fun li →
                    let tySi = List.assoc li fS in
                    let tyTi = List.assoc li fT in
                    (li, join tySi tyTi))
                 commonLabels in
      TyRecord(commonFields)
  | _ →
      TyTop
```

```
and meet tyS tyT =
  match (tyS,tyT) with
    (_,TyTop) →
      tyS
  | (TyTop,_) →
      tyT
  | (TyArr(tyS1,tyS2),TyArr(tyT1,tyT2)) →
      TyArr(join tyS1 tyT1, meet tyS2 tyT2)
  | (TyBool,TyBool) →
      TyBool
  | (TyRecord(fS), TyRecord(fT)) →
      let labelsS = List.map (fun (li,_) → li) fS in
      let labelsT = List.map (fun (li,_) → li) fT in
      let allLabels =
        List.append
          labelsS
          (List.find_all
            (fun l → not (List.mem l labelsS)) labelsT) in
      let allFields =
        List.map (fun li →
                    if List.mem li allLabels then
                      let tySi = List.assoc li fS in
                      let tyTi = List.assoc li fT in
                      (li, meet tySi tyTi)
                    else if List.mem li labelsS then
                      (li, List.assoc li fS)
                    else
                      (li, List.assoc li fT))
                 allLabels in
      TyRecord(allFields)
  | _ →
      raise Not_found

let rec typeof ctx t =
  match t with
    ...
  | TmTrue(fi) →
      TyBool
  | TmFalse(fi) →
      TyBool
  | TmIf(fi,t1,t2,t3) →
      if subtype (typeof ctx t1) TyBool then
        join (typeof ctx t2) (typeof ctx t3)
      else error fi "guard of conditional not a boolean"
```

解答 17.3.2. rcssubbot の実装を見よ。

解答 18.6.1.

```
DecCounter = {get:Unit→Nat, inc:Unit→Unit, reset:Unit→Unit,
              dec:Unit→Unit};

decCounterClass =
  λr:CounterRep.
    let super = resetCounterClass r in
      {get   = super.get,
       inc   = super.inc,
       reset = super.reset,
       dec   = λ_:Unit. r.x:=pred(!(r.x))};
```

解答 18.7.1.

```
BackupCounter2 = {get:Unit→Nat, inc:Unit→Unit,
                  reset:Unit→Unit, backup: Unit→Unit,
                  reset2:Unit→Unit, backup2: Unit→Unit};
BackupCounterRep2 = {x: Ref Nat, b: Ref Nat, b2: Ref Nat};

backupCounterClass2 =
  λr:BackupCounterRep2.
    let super = backupCounterClass r in
       {get = super.get, inc = super.inc,
        reset = super.reset, backup = super.backup,
        reset2 = λ_:Unit. r.x:=!(r.b2),
        backup2 = λ_:Unit. r.b2:=!(r.x)};
```

解答 18.11.1.

```
instrCounterClass =
  λr:InstrCounterRep.
     λself: Unit→InstrCounter.
       λ_:Unit.
         let super = setCounterClass r self unit in
            {get = λ_:Unit. (r.a:=succ(!(r.a)); super.get unit),
             set = λi:Nat. (r.a:=succ(!(r.a)); super.set i),
             inc = super.inc,
             accesses = λ_:Unit. !(r.a)};

ResetInstrCounter = {get:Unit→Nat, set:Nat→Unit,
                     inc:Unit→Unit, accesses:Unit→Nat,
                     reset:Unit→Unit};

resetInstrCounterClass =
  λr:InstrCounterRep.
     λself: Unit→ResetInstrCounter.
       λ_:Unit.
         let super = instrCounterClass r self unit in
            {get = super.get,
             set = super.set,
             inc = super.inc,
             accesses = super.accesses,
             reset = λ_:Unit. r.x:=0};
```

```
BackupInstrCounter = {get:Unit→Nat, set:Nat→Unit,
                      inc:Unit→Unit, accesses:Unit→Nat,
                      backup:Unit→Unit, reset:Unit→Unit};

BackupInstrCounterRep = {x: Ref Nat, a: Ref Nat, b: Ref Nat};

backupInstrCounterClass =
  λr:BackupInstrCounterRep.
    λself: Unit→BackupInstrCounter.
      λ_:Unit.
        let super = resetInstrCounterClass r self unit in
          {get = super.get,
           set = super.set,
           inc = super.inc,
           accesses = super.accesses,
           reset = λ_:Unit. r.x:=!(r.b),
           backup = λ_:Unit. r.b:=!(r.x)};

newBackupInstrCounter =
  λ_:Unit. let r = {x=ref 1, a=ref 0, b=ref 0} in
              fix (backupInstrCounterClass r) unit;
```

解答 18.13.1. 同一性検査を行う一つの方法は、参照セルを使う方法である。オブジェクトの内部表現を拡張して、型 Ref Nat のインスタンス変数 id を持たせる。

```
IdCounterRep = {x: Ref Nat, id: Ref (Ref Nat)};
```

また、id フィールドを返すだけの id メソッドも持たせる。

```
IdCounter = {get:Unit→Nat, inc:Unit→Unit, id:Unit→(Ref Nat)};
idCounterClass =
  λr:IdCounterRep.
    {get = λ_:Unit. !(r.x),
     inc = λ_:Unit. r.x:=succ(!(r.x)),
     id  = λ_:Unit. !(r.id)};
```

ここで、sameObject 関数は id メソッドを持つ二つのオブジェクトを受け取り、その id メソッドが返す参照が一致するかどうかを検査する。

```
sameObject =
  λa:{id:Unit→(Ref Nat)}. λb:{id:Unit→(Ref Nat)}.
    ((b.id unit) := 1;
     (a.id unit) := 0;
     iszero (!(b.id unit)));
```

ここでのトリックは、別名付けを利用して二つの参照セルが同じかどうかを検査することである。二つ目が 0 以外の値を持つようにしてから一つ目に 0 を代入し、二つ目が 0 になっているかどうかを検査する。

解答 19.4.1. すべてのクラス宣言は `extends` 節を持たなければならず、これらの節は循環することが許されないので、どのクラスから `extends` 節を辿っていっても必ず `Object` で終わることになる。

解答 19.4.2. 明らかな改良としては、キャストの三つの型付け規則を一つにまとめ、

$$\frac{\Gamma \vdash \mathtt{t}_0 : \mathtt{D}}{\Gamma \vdash (\mathtt{C})\mathtt{t}_0 : \mathtt{C}} \qquad \text{(T-CAST)}$$

愚かなキャストの概念を捨てることが考えられる。他には、コンストラクタは結局何もやらないので、消してしまうことも考えられる。

解答 19.4.6.

(1) FJ に対するインターフェイスの形式化は定型作業である。

(2) 以下のインターフェイスを宣言したとする。

```
interface A {}
interface B {}
interface C extends A,B {}
interface D extends A,B {}
```

C と D は A と B の両方を共通の上界として持つが、上限は持たない。

(3) 条件式に対する標準的なアルゴリズム的規則は以下であった。

$$\frac{\Gamma \vdash \mathtt{t}_1 : \mathtt{boolean} \qquad \Gamma \vdash \mathtt{t}_2 : \mathtt{E}_2 \qquad \Gamma \vdash \mathtt{t}_3 : \mathtt{E}_3}{\Gamma \vdash \mathtt{t}_1 \mathbin{?} \mathtt{t}_2 : \mathtt{t}_3 : \mathtt{E}_2 \vee \mathtt{E}_3}$$

Java ではこれの代わりに、以下のように制限した規則を用いる。

$$\frac{\Gamma \vdash \mathtt{t}_1 : \mathtt{boolean} \qquad \Gamma \vdash \mathtt{t}_2 : \mathtt{E}_2 \qquad \Gamma \vdash \mathtt{t}_3 : \mathtt{E}_3 \qquad \Gamma \vdash \mathtt{E}_2 <: \mathtt{E}_3}{\Gamma \vdash \mathtt{t}_1 \mathbin{?} \mathtt{t}_2 : \mathtt{t}_3 : \mathtt{E}_3}$$

$$\frac{\Gamma \vdash \mathtt{t}_1 : \mathtt{boolean} \qquad \Gamma \vdash \mathtt{t}_2 : \mathtt{E}_2 \qquad \Gamma \vdash \mathtt{t}_3 : \mathtt{E}_3 \qquad \Gamma \vdash \mathtt{E}_3 <: \mathtt{E}_2}{\Gamma \vdash \mathtt{t}_1 \mathbin{?} \mathtt{t}_2 : \mathtt{t}_3 : \mathtt{E}_2}$$

これらの規則は直観的には健全だが、FJ で用いられる小ステップスタイルの操作的意味論と相性がよくない。実際、型の保存の性質が成り立たなくなってしまう（これを示す例を作るのは難しくない）。

解答 19.4.7. 驚くべきことに、`super` を扱うのは `self` を扱うより難しい。これは、「現在実行しているメソッド本体」が属するクラスを何らかの方法で覚えておかなければならないからである。これを行うには少なくとも二つの方法がある。

(1) `super` の参照がどこを見に行くべきかについて、項に何らかの指示を注釈しておく方法。

(2) クラス表全体を書き換えるプリプロセスのステップを追加して、`super` の参照を `this` の参照に置き換え、さらにどのクラスのメソッドかを示す情報を呼び出すメソッド名に埋め込む（mangle する）方法。

解答 19.5.1. 本体の証明を与える前に、必要な補題を幾つか示す。いつものように、重要な補題 A.14 は型付けと代入に関連するものである。

補題 A.13. $mtype(\mathtt{m},\mathtt{D}) = \overline{\mathtt{C}}\to\mathtt{C}_0$ ならば、すべての $\mathtt{C} <: \mathtt{D}$ に対して $mtype(\mathtt{m},\mathtt{C}) = \overline{\mathtt{C}}\to\mathtt{C}_0$ □

[証明] $\mathtt{C} <: \mathtt{D}$ の導出に関する単純な帰納法。m が $CT(\mathtt{C})$ に定義されているかされていないかにかかわらず、$CT(\mathtt{C})$ = `class C extends E {...}` のとき、$mtype(\mathtt{m},\mathtt{C})$ は $mtype(\mathtt{m},\mathtt{E})$ と同じになるはずであることに注意されたい。 □

補題 A.14.［項の代入は型付けを保存］ $\Gamma, \overline{\mathtt{x}}:\overline{\mathtt{B}} \vdash \mathtt{t} : \mathtt{D}$ かつ $\Gamma \vdash \overline{\mathtt{s}} : \overline{\mathtt{A}}$（ただし $\overline{\mathtt{A}} <: \overline{\mathtt{B}}$）ならば、ある $\mathtt{C} <: \mathtt{D}$ が存在して $\Gamma \vdash [\overline{\mathtt{x}} \mapsto \overline{\mathtt{s}}]\mathtt{t} : \mathtt{C}$ □

[証明] $\Gamma, \overline{\mathtt{x}}:\overline{\mathtt{B}} \vdash \mathtt{t} : \mathtt{D}$ の導出に関する帰納法による。ここでの直観は、部分型付けのあるラムダ計算と全く同じである（当然詳細は少し変わる）。最も興味深いのは最後の二つの場合である。

T-VAR の場合　$\mathtt{t} = \mathtt{x}$　$\mathtt{x}:\mathtt{D} \in \Gamma$

$\mathtt{x} \notin \overline{\mathtt{x}}$ ならば、$[\overline{\mathtt{x}} \mapsto \overline{\mathtt{s}}]\mathtt{x} = \mathtt{x}$ なので結論は自明である。一方 $\mathtt{x} = \mathtt{x}_i$ かつ $\mathtt{D} = \mathtt{B}_i$ ならば、$[\overline{\mathtt{x}} \mapsto \overline{\mathtt{s}}]\mathtt{x} = \mathtt{s}_i$ なので、$\mathtt{C} = \mathtt{A}_i$ とおけばこの場合も終了する。

T-FIELD の場合　$\mathtt{t} = \mathtt{t}_0.\mathtt{f}_i$　$\Gamma, \overline{\mathtt{x}}:\overline{\mathtt{B}} \vdash \mathtt{t}_0 : \mathtt{D}_0$　$fields(\mathtt{D}_0) = \overline{\mathtt{C}}\ \overline{\mathtt{f}}$　$\mathtt{D} = \mathtt{C}_i$

帰納法の仮定により、ある $\mathtt{C}_0$ が存在して $\Gamma \vdash [\overline{\mathtt{x}} \mapsto \overline{\mathtt{s}}]\mathtt{t}_0 : \mathtt{C}_0$ かつ $\mathtt{C}_0 <: \mathtt{D}_0$。ある $\overline{\mathtt{D}}\ \overline{\mathtt{g}}$ が存在して $fields(\mathtt{C}_0) = (fields(\mathtt{D}_0), \overline{\mathtt{D}}\ \overline{\mathtt{g}})$ となることは簡単に確認できる。ゆえに、T-FIELD より、$\Gamma \vdash ([\overline{\mathtt{x}} \mapsto \overline{\mathtt{s}}]\mathtt{t}_0).\mathtt{f}_i : \mathtt{C}_i$。

T-INVK の場合　$\mathtt{t} = \mathtt{t}_0.\mathtt{m}(\overline{\mathtt{t}})$　$\Gamma, \overline{\mathtt{x}}:\overline{\mathtt{B}} \vdash \mathtt{t}_0 : \mathtt{D}_0$　$mtype(\mathtt{m},\mathtt{D}_0) = \overline{\mathtt{E}}\to\mathtt{D}$
$\Gamma, \overline{\mathtt{x}}:\overline{\mathtt{B}} \vdash \overline{\mathtt{t}} : \overline{\mathtt{D}}$　$\overline{\mathtt{D}} <: \overline{\mathtt{E}}$

帰納法の仮定により、ある $\mathtt{C}_0$ と $\overline{\mathtt{C}}$ が存在して

$$\Gamma \vdash [\overline{\mathtt{x}} \mapsto \overline{\mathtt{s}}]\mathtt{t}_0 : \mathtt{C}_0 \qquad \mathtt{C}_0 <: \mathtt{D}_0 \qquad \Gamma \vdash [\overline{\mathtt{x}} \mapsto \overline{\mathtt{s}}]\overline{\mathtt{t}} : \overline{\mathtt{C}} \qquad \overline{\mathtt{C}} <: \overline{\mathtt{D}}$$

である。補題 A.13 より、$mtype(\mathtt{m},\mathtt{C}_0) = \overline{\mathtt{E}}\to\mathtt{D}$。さらに <: の推移律により $\overline{\mathtt{C}} <: \overline{\mathtt{E}}$。したがって、T-INVK より、$\Gamma \vdash [\overline{\mathtt{x}} \mapsto \overline{\mathtt{s}}]\mathtt{t}_0.\mathtt{m}([\overline{\mathtt{x}} \mapsto \overline{\mathtt{s}}]\overline{\mathtt{t}}) : \mathtt{D}$。

T-NEW の場合　$\mathtt{t} = \mathtt{new\ D}(\overline{\mathtt{t}})$　$fields(\mathtt{D}) = \overline{\mathtt{D}}\ \overline{\mathtt{f}}$　$\Gamma, \overline{\mathtt{x}}:\overline{\mathtt{B}} \vdash \overline{\mathtt{t}} : \overline{\mathtt{C}}$　$\overline{\mathtt{C}} <: \overline{\mathtt{D}}$

帰納法の仮定により、$\overline{\mathtt{E}} <: \overline{\mathtt{C}}$ なる $\overline{\mathtt{E}}$ が存在して $\Gamma \vdash [\overline{\mathtt{x}} \mapsto \overline{\mathtt{s}}]\overline{\mathtt{t}} : \overline{\mathtt{E}}$。<: の推移律から $\overline{\mathtt{E}} <: \overline{\mathtt{D}}$ とわかる。したがって、T-NEW より、$\Gamma \vdash \mathtt{new\ D}([\overline{\mathtt{x}} \mapsto \overline{\mathtt{s}}]\overline{\mathtt{t}}) : \mathtt{D}$。

T-UCAST の場合　$\mathtt{t} = (\mathtt{D})\mathtt{t}_0$　$\Gamma, \overline{\mathtt{x}}:\overline{\mathtt{B}} \vdash \mathtt{t}_0 : \mathtt{C}$　$\mathtt{C} <: \mathtt{D}$

帰納法の仮定により、ある E が存在して $\Gamma \vdash [\overline{\mathtt{x}} \mapsto \overline{\mathtt{s}}]\mathtt{t}_0 : \mathtt{E}$ かつ $\mathtt{E} <: \mathtt{C}$。<: の推移律から $\mathtt{E} <: \mathtt{D}$ とわかる。ここから T-UCAST により $\Gamma \vdash (\mathtt{D})([\overline{\mathtt{x}} \mapsto \overline{\mathtt{s}}]\mathtt{t}_0) : \mathtt{D}$ を得る。

T-DCAST の場合　$\mathtt{t} = (\mathtt{D})\mathtt{t}_0$　$\Gamma, \overline{\mathtt{x}}:\overline{\mathtt{B}} \vdash \mathtt{t}_0 : \mathtt{C}$　$\mathtt{D} <: \mathtt{C}$　$\mathtt{D} \neq \mathtt{C}$

帰納法の仮定により、ある E が存在して $\Gamma \vdash [\overline{\mathtt{x}} \mapsto \overline{\mathtt{s}}]\mathtt{t}_0 : \mathtt{E}$ かつ $\mathtt{E} <: \mathtt{C}$。$\mathtt{E} <: \mathtt{D}$ または $\mathtt{D} <: \mathtt{E}$ ならば、それぞれ T-UCAST か T-DCAST により、$\Gamma \vdash (\mathtt{D})([\overline{\mathtt{x}} \mapsto \overline{\mathtt{s}}]\mathtt{t}_0) : \mathtt{D}$。一方 $\mathtt{D} \not<: \mathtt{E}$ かつ $\mathtt{E} \not<: \mathtt{D}$ ならば、T-SCAST より $\Gamma \vdash (\mathtt{D})([\overline{\mathtt{x}} \mapsto \overline{\mathtt{s}}]\mathtt{t}_0) : \mathtt{D}$（ただし愚かさの警告が出る）。

T-SCAST の場合　$\mathtt{t} = (\mathtt{D})\mathtt{t}_0$　$\Gamma, \overline{\mathtt{x}}:\overline{\mathtt{B}} \vdash \mathtt{t}_0 : \mathtt{C}$　$\mathtt{D} \not<: \mathtt{C}$　$\mathtt{C} \not<: \mathtt{D}$

帰納法の仮定により、ある E が存在して $\Gamma \vdash [\overline{\mathtt{x}} \mapsto \overline{\mathtt{s}}]\mathtt{t}_0 : \mathtt{E}$ かつ $\mathtt{E} <: \mathtt{C}$。これは $\mathtt{E} \not<: \mathtt{D}$ を意味する（これを理解するには、FJ の各クラスがちょうど一つのスーパークラスを持つことに注目せよ。$\mathtt{E} <: \mathtt{C}$ かつ $\mathtt{E} <: \mathtt{D}$ ならば、$\mathtt{C} <: \mathtt{D}$ または $\mathtt{D} <: \mathtt{C}$ であることは導かれる）。よって T-SCAST より、$\Gamma \vdash (\mathtt{D})([\overline{\mathtt{x}} \mapsto \overline{\mathtt{s}}]\mathtt{t}_0) : \mathtt{D}$（ただし愚かさの警告が出る）。 □

補題 A.15.［弱化］　$\Gamma \vdash \mathtt{t} : \mathtt{C}$ ならば $\Gamma, \mathtt{x}:\mathtt{D} \vdash \mathtt{t} : \mathtt{C}$ □

［証明］　単純な帰納法。 □

補題 A.16.　$\mathit{mtype}(\mathtt{m}, \mathtt{C_0}) = \overline{\mathtt{D}} \to \mathtt{D}$、かつ $\mathit{mbody}(\mathtt{m}, \mathtt{C_0}) = (\overline{\mathtt{x}}, \mathtt{t})$ ならば、ある $\mathtt{D_0}$ と $\mathtt{C} <: \mathtt{D}$ が存在して $\mathtt{C_0} <: \mathtt{D_0}$ かつ $\overline{\mathtt{x}} : \overline{\mathtt{D}}, \mathtt{this} : \mathtt{D_0} \vdash \mathtt{t} : \mathtt{C}$ □

［証明］　$\mathit{mbody}(\mathtt{m}, \mathtt{C_0})$ の導出に関する帰納法による。基底ケース（m が $\mathtt{C_0}$ で定義される場合）は、m が $\mathit{CT}(\mathtt{C_0})$ で定義されること、そしてクラス表が正しい形式を持つので、T-Method によって $\overline{\mathtt{x}} : \overline{\mathtt{D}}, \mathtt{this} : \mathtt{C_0} \vdash \mathtt{t} : \mathtt{C}$ を導出したはずであることから、容易である。帰納法のステップも単純である。 □

これで、型安全性の定理の証明を行う準備が整った。

［定理 19.5.1 の証明］　$\mathtt{t} \longrightarrow \mathtt{t}'$ の導出に関する帰納法で、最後の規則で場合分けする。T-DCast の場合（最後から二番目）で、愚かさの警告がどのように生成されるかに注目せよ。

E-ProjNew の場合　$\mathtt{t} = \mathtt{new\ C_0(\overline{v}).f_i}$　　$\mathtt{t}' = \mathtt{v_i}$　　$\mathit{fields}(\mathtt{C_0}) = \overline{\mathtt{D}}\ \overline{\mathtt{f}}$

t の形から、$\Gamma \vdash \mathtt{t} : \mathtt{C}$ の導出の最後の規則は T-Field だとわかる。その前提はある $\mathtt{D_0}$ に対して $\Gamma \vdash \mathtt{new\ C_0(\overline{v})} : \mathtt{D_0}$ である。t の形からは、$\mathtt{C} = \mathtt{D_i}$ もわかる。同様に、$\Gamma \vdash \mathtt{new\ C_0(\overline{v})} : \mathtt{D_0}$ の導出の最後の規則は T-New でなければならない。ここから、その前提として $\Gamma \vdash \overline{\mathtt{v}} : \overline{\mathtt{C}}$ かつ $\overline{\mathtt{C}} <: \overline{\mathtt{D}}$ とわかり、さらに、$\mathtt{D_0} = \mathtt{C_0}$ とわかる。特に $\Gamma \vdash \mathtt{v_i} : \mathtt{C_i}$ なので、$\mathtt{C_i} <: \mathtt{D_i}$ であることからこの場合は終了する。

E-InvkNew の場合　$\mathtt{t} = \mathtt{(new\ C_0(\overline{v})).m(\overline{u})}$　　$\mathtt{t}' = [\overline{\mathtt{u}}/\overline{\mathtt{x}}, \mathtt{new\ C_0(\overline{v})/this}]\mathtt{t_0}$
$\mathit{mbody}(\mathtt{m}, \mathtt{C_0}) = (\overline{\mathtt{x}}, \mathtt{t_0})$

$\Gamma \vdash \mathtt{t} : \mathtt{C}$ の導出の最後の規則は T-Invk と T-New である。その前提は以下である。

$$\Gamma \vdash \mathtt{new\ C_0(\overline{v})} : \mathtt{C_0} \qquad \Gamma \vdash \overline{\mathtt{u}} : \overline{\mathtt{C}} \qquad \overline{\mathtt{C}} <: \overline{\mathtt{D}} \qquad \mathit{mtype}(\mathtt{m}, \mathtt{C_0}) = \overline{\mathtt{D}} \to \mathtt{C}$$

補題 A.16 より、$\mathtt{C_0} <: \mathtt{D_0}$ かつ $\mathtt{B} <: \mathtt{C}$ であるような $\mathtt{D_0}$ と B が存在して $\overline{\mathtt{x}} : \overline{\mathtt{D}}, \mathtt{this} : \mathtt{D_0} \vdash \mathtt{t_0} : \mathtt{B}$。補題 A.15 より、$\Gamma, \overline{\mathtt{x}} : \overline{\mathtt{D}}, \mathtt{this} : \mathtt{D_0} \vdash \mathtt{t_0} : \mathtt{B}$。よって、補題 A.14 より、ある $\mathtt{E} <: \mathtt{B}$ が存在して $\Gamma \vdash [\overline{\mathtt{x}} \mapsto \overline{\mathtt{u}}, \mathtt{this} \mapsto \mathtt{new\ C_0(\overline{v})}]\mathtt{t_0} : \mathtt{E}$ となる。<: の推移律から、$\mathtt{E} <: \mathtt{C}$ を得る。$\mathtt{C}' = \mathtt{E}$ とおけばこの場合は完了する。

E-CastNew の場合　$\mathtt{t} = \mathtt{(D)(new\ C_0(\overline{v}))}$　　$\mathtt{C_0} <: \mathtt{D}$　　$\mathtt{t}' = \mathtt{new\ C_0(\overline{v})}$

$\Gamma \vdash \mathtt{(D)(new\ C_0(\overline{v}))} : \mathtt{C}$ の証明は T-UCast で終わらなければならない。なぜなら、T-SCast または T-DCast で終わっているとすると仮定 $\mathtt{C_0} <: \mathtt{D}$ と矛盾するためである。T-UCast の前提から、$\Gamma \vdash \mathtt{new\ C_0(\overline{v})} : \mathtt{C_0}$ かつ $\mathtt{D} = \mathtt{C}$ とわかり、この場合は終了する。

合同規則の場合はどれも簡単である。一つだけ証明する。

E-Cast の場合　$\mathtt{t} = \mathtt{(D)t_0}$　　$\mathtt{t}' = \mathtt{(D)t_0'}$　　$\mathtt{t_0} \longrightarrow \mathtt{t_0'}$

最後に用いられている型付け規則により、さらに三つに場合分けできる。

T-UCast の場合　$\Gamma \vdash \mathtt{t_0} : \mathtt{C_0}$　　$\mathtt{C_0} <: \mathtt{D}$　　$\mathtt{D} = \mathtt{C}$

帰納法の仮定により、ある $\mathtt{C_0'} <: \mathtt{C_0}$ が存在して $\Gamma \vdash \mathtt{t_0'} : \mathtt{C_0'}$。<: の推移律から、$\mathtt{C_0'} <: \mathtt{C}$。したがって、T-UCast より、$\Gamma \vdash \mathtt{(C)t_0'} : \mathtt{C}$（追加の愚かさの警告は出ない）。

T-DCast の場合　$\Gamma \vdash \mathtt{t_0} : \mathtt{C_0}$　　$\mathtt{D} <: \mathtt{C_0}$　　$\mathtt{D} = \mathtt{C}$

帰納法の仮定により、ある $\mathtt{C_0'} <: \mathtt{C_0}$ が存在して $\Gamma \vdash \mathtt{t_0'} : \mathtt{C_0'}$。$\mathtt{C_0'} <: \mathtt{C}$ または $\mathtt{C} <: \mathtt{C_0'}$ ならば、T-UCast か T-DCast により $\Gamma \vdash \mathtt{(C)t_0'} : \mathtt{C}$（追加の愚かさの警告は出ない）。一方 $\mathtt{C_0'} \not<: \mathtt{C}$ かつ $\mathtt{C} \not<: \mathtt{C_0'}$ ならば、T-SCast により $\Gamma \vdash \mathtt{(C)t_0'} : \mathtt{C}$（ただし愚かさの警告が出る）。

T-SCAST の場合 $\Gamma \vdash t_0 : C_0 \quad D \not<: C_0 \quad C_0 \not<: D \quad D = C$

帰納法の仮定により、ある $C'_0 <: C_0$ が存在して $\Gamma \vdash t'_0 : C'_0$。よって、$C'_0 \not<: C$ と $C \not<: C'_0$ が共に成り立つ。したがって $\Gamma \vdash (C)t'_0 : C$ (ただし愚かさの警告が出る)。 □

解答 20.1.1.

```
  Tree = μX. <leaf:Unit, node:{Nat,X,X}>;
  leaf = <leaf=unit> as Tree;
▶ leaf : Tree

  node = λn:Nat. λt₁:Tree. λt₂:Tree. <node={n,t₁,t₂}> as Tree;
▶ node : Nat → Tree → Tree → Tree

  isleaf = λl:Tree. case l of <leaf=u> ⇒ true | <node=p> ⇒ false;
▶ isleaf : Tree → Bool

  label = λl:Tree. case l of <leaf=u> ⇒ 0 | <node=p> ⇒ p.1;
▶ label : Tree → Nat

  left = λl:Tree. case l of <leaf=u> ⇒ leaf | <node=p> ⇒ p.2;
▶ left : Tree → Tree

  right = λl:Tree. case l of <leaf=u> ⇒ leaf | <node=p> ⇒ p.3;
▶ right : Tree → Tree

  append = fix (λf:NatList→NatList→NatList.
                  λl1:NatList. λl2:NatList.
                  if isnil l1 then l2 else
                  cons (hd l1) (f (tl l1) l2));
▶ append : NatList → NatList → NatList

  preorder = fix (λf:Tree→NatList. λt:Tree.
                    if isleaf t then nil else
                    cons (label t)
                         (append (f (left t)) (f (right t))));
▶ preorder : Tree → NatList

  t1 = node 1 leaf leaf;
  t2 = node 2 leaf leaf;
  t3 = node 3 t1 t2;
  t4 = node 4 t3 t3;
  l = preorder t4;
  hd l;
▶ 4 : Nat

  hd (tl l);
▶ 3 : Nat

  hd (tl (tl l));
▶ 1 : Nat
```

解答 20.1.2.

```
fib = fix (λf: Nat→Nat→Stream. λm:Nat. λn:Nat. λ_:Unit.
          {n, f n (plus m n)}) 0 1;
▶ fib : Stream
```

解答 20.1.3.

```
Counter = μC. {get:Nat, inc:Unit→C, dec:Unit→C,
               reset:Unit→C, backup:Unit→C};
c = let create =
       fix (λcr: {x:Nat,b:Nat}→Counter. λs: {x:Nat,b:Nat}.
              {get    = s.x,
               inc    = λ_:Unit. cr {x=succ(s.x),b=s.b},
               dec    = λ_:Unit. cr {x=pred(s.x),b=s.b},
               backup = λ_:Unit. cr {x=s.x,b=s.x},
               reset  = λ_:Unit. cr {x=s.b,b=s.b}    })
    in create {x=0,b=0};
▶ c : Counter
```

解答 20.1.4.

```
D = μX. <nat:Nat, bool:Bool, fn:X→X>;

lam = λf:D→D. <fn=f> as D;
ap = λf:D. λa:D.
       case f of
         <nat=n> ⇒ divergeD unit
       | <bool=b> ⇒ divergeD unit
       | <fn=f> ⇒ f a;
ifd = λb:D. λt:D. λe:D.
        case b of
          <nat=n> ⇒ divergeD unit
        | <bool=b> ⇒ (if b then t else e)
        | <fn=f> ⇒ divergeD unit;
tru = <bool=true> as D;
fls = <bool=false> as D;
ifd fls one zro;
▶ <nat=0> as D : D

ifd fls one fls;
▶ <bool=false> as D : D
```

正しく型付けできない項をコード化できてしまうことに懸念を感じる読者は、ここで行っていることが、型無しの項という対象言語を、再帰型を持つ単純型付きラムダ計算というメタ言語で表現する**データ構造**を構築していることに着目せよ。これができるという事実には、さまざまな型付きもしくは型無しのラムダ計算の項を ML のデータ構造として表現できる、という（本書の実装の章を通して使ってきた）事実以上の意外性はない。

解答 20.1.5.

```
lam = λf:D→D. <fn=f> as D;
ap = λf:D. λa:D. case f of
                    <nat=n> ⇒ diverge_D unit
                  | <fn=f> ⇒ f a
                  | <rcd=r> ⇒ diverge_D unit;
rcd = λfields:Nat→D. <rcd=fields> as D;
prj = λf:D. λn:Nat. case f of
                    <nat=n> ⇒ diverge_D unit
                  | <fn=f> ⇒ diverge_D unit
                  | <rcd=r> ⇒ r n;
myrcd = rcd (λn:Nat. if iszero 0 then zro
                     else if iszero (pred n) then one
                     else diverge_D unit);
```

解答 20.2.1. 以下は、同型再帰形式で表現した、幾つかのより興味深い例である。

```
Hungry = μA. Nat → A;
f = fix (λf: Nat→Hungry. λn:Nat. fold [Hungry] f);
ff = fold [Hungry] f;
ff1 = (unfold [Hungry] ff) 0;
ff2 = (unfold [Hungry] ff1) 2;

fix_T =
  λf:T→T.
    (λx:(μA.A→T). f ((unfold [μA.A→T] x) x))
    (fold [μA.A→T] (λx:(μA.A→T). f ((unfold [μA.A→T] x) x)));

D = μX. X→X;
lam = λf:D→D. fold [D] f;
ap = λf:D. λa:D. (unfold [D] f) a;

Counter = μC. {get:Nat, inc:Unit→C};
c = let create = fix (λcr: {x:Nat}→Counter. λs: {x:Nat}.
                      fold [Counter]
                        {get = s.x,
                         inc = λ_:Unit. cr {x=succ(s.x)}})
    in create {x=0};
c1 = (unfold [Counter] c).inc unit;
(unfold [Counter] c1).get;
```

解答 21.1.7.

$$
\begin{array}{lclclcl}
E_2(\varnothing) & = & \{a\} & \quad & E_2(\{a,b\}) & = & \{a,c\} \\
E_2(\{a\}) & = & \{a\} & & E_2(\{a,c\}) & = & \{a,b\} \\
E_2(\{b\}) & = & \{a\} & & E_2(\{b,c\}) & = & \{a,b\} \\
E_2(\{c\}) & = & \{a,b\} & & E_2(\{a,b,c\}) & = & \{a,b,c\}
\end{array}
$$

E_2 について閉じている集合は $\{a\}$ および $\{a,b,c\}$ である。E_2 について整合的である集合は $\varnothing$、$\{a\}$、$\{a,b,c\}$ である。E_2 の最小不動点は $\{a\}$ であり、最大不動点は $\{a,b,c\}$ である。

解答 21.1.9. 自然数上の通常の帰納法の原理は、以下の流れで証明できる。まず、生成関数 $F \in \mathcal{P}(\mathbb{N}) \to \mathcal{P}(\mathbb{N})$ を

$$F(X) \;=\; \{0\} \cup \{i+1 \mid i \in X\}$$

で定義する。次に、述語（すなわち自然数の集合）P が、$P(0)$ であり、かつ $P(i)$ ならば $P(i+1)$ であるとする。すると、F の定義より、$F(P) \subseteq P$ である（すなわち P が F について閉じている）ことが容易に確認できる。帰納法の原理より $\mu F \subseteq P$ である。しかし、μF は自然数の全体なので（実際、このことを自然数の集合の**定義**とみなすこともできる)、すべての $n \in \mathbb{N}$ について $P(n)$ が成り立つ。

辞書式帰納法の場合には、$F \in \mathcal{P}(\mathbb{N} \times \mathbb{N}) \to \mathcal{P}(\mathbb{N} \times \mathbb{N})$ を

$$F(X) \;=\; \{(m,n) \mid \forall (m',n') < (m,n),\ (m',n') \in X\}$$

として定義する。そして、述語（すなわち数の組の集合）P で、すべての $(m',n') < (m,n)$ について $P(m',n')$ であるなら、$P(m,n)$ も成り立つものが与えられたとする。以前と同様に、F の定義から $F(P) \subseteq P$ である（すなわち P が F について閉じている）ことは容易に確認できる。帰納法の原理より $\mu F \subseteq P$ 。証明を完了するには、あとは μF が実際にすべての数の組からなる集合であることを確認しなくてはならない（これがこの議論の唯一のややこしい点である)。このことは二段階に分けて論じられる。第一に、$\mathbb{N} \times \mathbb{N}$ は F について閉じている（これは F の定義から直ちにいえる)。第二に、$\mathbb{N} \times \mathbb{N}$ の真部分集合であって F について閉じているものは存在しない（すなわち、$\mathbb{N} \times \mathbb{N}$ は F について閉じた最小の集合である）ということを示す。このことを確認するために、F について閉じている、より小さな集合 Y があったとし、(m,n) を Y に**含まれない**最小の二つ組とする。すると、F の定義から、$F(Y) \not\subseteq Y$ である（すなわち Y は F について閉じていない）ことになるが、これは矛盾である。

解答 21.2.2. 木を、以下の制約を満たす、部分関数 $T \in \{1,2\}^* \rightharpoonup \{\to, \times, \mathsf{Top}\}$ として定義する。

- $T(\bullet)$ は定義されている。
- $T(\pi,\sigma)$ が定義されているならば、$T(\pi)$ も定義されている。

ここで木のノードにおける $\to, \times,$Top の出現には制約が全くないことに注意されたい。例えば、Top を持つノードが非自明な子を持つなどしてもよい。21.2 節での場合と同様に、記号 $\to$, $\times$, Top を木の上の演算子としても多重定義する。

すべての木からなる集合を普遍集合 $\mathcal{U}$ とする。生成関数 F を馴染み深い型の文法に基づく以下の関数とする。

$$\begin{array}{rcl} F(X) & = & \{\mathsf{Top}\} \\ & \cup & \{T_1 \times T_2 \mid T_1, T_2 \in X\} \\ & \cup & \{T_1 \to T_2 \mid T_1, T_2 \in X\} \end{array}$$

$\mathcal{T}$ および $\mathcal{U}$ の定義から、$\mathcal{T} \subseteq \mathcal{U}$ であることがわかる。そのため、興味のある等式 $\mathcal{T} = \nu F$ および $\mathcal{T}_f = \mu F$ 中の集合を比較することには意味がある。残るはこれらの等式が真であることを確認するだけである。

$\mathcal{T} \subseteq \nu F$ は、$\mathcal{T}$ が F について整合的であるという事実から、余帰納法の原理を用いて示せる。$\nu F \subseteq \mathcal{T}$ を示すには、任意の $T \in \nu F$ について、定義 21.2.1 の最後の二条件が満たされていることを確認すればよい。これは π の長さに関する帰納法で示せる。

$\mu F \subseteq \mathcal{T}_f$ は、$\mathcal{T}_f$ が F について閉じているという事実から、帰納法の原理を用いて示せる。$\mathcal{T}_f \subseteq \mu F$ を得るためには、T のサイズに関する帰納法を用いて、$\mathsf{T} \in \mathcal{T}_f$ ならば $\mathsf{T} \in \mu F$ であることを示す。

(T ∈ $\mathcal{T}_f$ のサイズは、$\pi \in \{1,2\}^*$ のうち $\mathsf{T}(\pi)$ が定義されている最長の π の長さとして定義できる。)

解答 21.3.3. 二つ組 (Top, Top × Top) は νS には含まれない。このことを確認するには単に、S の定義から、どんな X についても、この二つ組が $S(X)$ に含まれないことを観察すればよい。したがって、S について整合的な集合で、この二つ組を含むものは存在しない。特に、νS は、(S について整合的な集合であるので) この二つ組を含まない。

解答 21.3.4. νS では関係しているものの、μS では関係していない木型の二つ組の例として、任意の無限型 T についての二つ組 (T, T) が挙げられる。二つ組の集合 $R = \{(\mathsf{Q},\mathsf{Q}) \mid \mathsf{Q}$ は T の部分木$\}$ を考える。S の定義を確認すれば、$R \subseteq S(R)$ であることが容易に得られ、余帰納法の原理を適用すれば、$R \subseteq \nu S$ が得られる。ここで $(\mathsf{T},\mathsf{T}) \in R$ なので、$(\mathsf{T},\mathsf{T}) \in \nu S$ である。その一方、μS は有限型だけを関係させるため、$(\mathsf{T},\mathsf{T}) \notin \mu S$ である[†9]。μS が有限の型だけを関係させることを証明するには、R' をすべての有限型の二つ組の集合とし、帰納法の原理により $\mu S \subseteq R'$ を導けばよい。

有限型の二つ組 (S, T) で、νS_f では関係しているが、μS_f では関係しないというものは存在しない。なぜなら、これら二つの不動点は一致するためである。このことは、任意の $\mathsf{S},\mathsf{T} \in \mathcal{T}_f$ について、$(\mathsf{S},\mathsf{T}) \in \nu S_f$ ならば $(\mathsf{S},\mathsf{T}) \in \mu S_f$ である、という事実から導かれる。(ではその事実は何から導かれるかというと、S と T は共に有限木であるから、S と T のサイズの和に関する帰納法によって得られる。T が Top の場合、$\mathsf{T}_1 \times \mathsf{T}_2$ の場合、$\mathsf{T}_1 \to \mathsf{T}_2$ の場合のそれぞれについて、S_f の定義を確認し、等式 $S_f(\nu S_f) = \nu S_f$ と等式 $S_f(\mu S_f) = \mu S_f$ を用いる。)

解答 21.3.8. まず、木型の上の同一関係を $I = \{(\mathsf{T},\mathsf{T}) \mid \mathsf{T} \in \mathcal{T}\}$ と定義する。もし、I が S について整合的であることを示せれば、余帰納法の原理により、$I \subseteq \nu S$ であること、すなわち νS が反射的であることがわかる。I が S について整合的であることを示すには、要素 $(\mathsf{T},\mathsf{T}) \in I$ を考え、T の形で場合分けすればよい。まず、T = Top とする。すると、(T, T) = (Top, Top) であり、定義により $S(I)$ に含まれる。次に、$\mathsf{T} = \mathsf{T}_1 \times \mathsf{T}_2$ とする。すると、$(\mathsf{T}_1,\mathsf{T}_1), (\mathsf{T}_2,\mathsf{T}_2) \in I$ であるので、S の定義から $(\mathsf{T}_1 \times \mathsf{T}_2, \mathsf{T}_1 \times \mathsf{T}_2) \in S(I)$ がいえる。$\mathsf{T} = \mathsf{T}_1 \to \mathsf{T}_2$ についても同様である。

解答 21.4.2. 余帰納法の原理により、$\mathcal{U} \times \mathcal{U}$ が F^{TR} について整合的であること ($\mathcal{U} \times \mathcal{U} \subseteq F^{TR}(\mathcal{U} \times \mathcal{U})$) を示せば十分である。$(x,y) \in \mathcal{U} \times \mathcal{U}$ とする。任意の $z \in \mathcal{U}$ を選択すれば、$(x,z), (z,y) \in \mathcal{U} \times \mathcal{U}$ であり、F^{TR} の定義により、$(x,y) \in F^{TR}(\mathcal{U} \times \mathcal{U})$ も成り立つ。

解答 21.5.2. 可逆性を確認するには、単に S_f と S の定義を調査し、各集合 $G_{(\mathsf{S},\mathsf{T})}$ が極小な要素を高々一つしか含まないことを確認すればよい。

S_f と S の定義の各節は、支持可能な要素の形と支持集合の中身を明示しており、$support_{S_f}$ と $support_S$ とを書き下すのは容易い。(定義 21.8.4 における S_m の支持関数と比較せよ。)

解答 21.5.4.

$$\frac{i \quad a}{h} \qquad \frac{b \quad c}{a} \qquad \frac{b}{d} \qquad \frac{d}{e} \qquad \frac{e}{b} \qquad \frac{f \quad g}{c} \qquad \frac{g}{f} \qquad \frac{}{g}$$

解答 21.5.6. 正しくない。$x \in \nu F \setminus \mu F$ は、支持グラフ内で循環に到達可能だとは限らない。循環ではなく無限列でもよい。例えば、$F(X) = \{0\} \cup \{n \mid n+1 \in X\}$ で定義される $F \in \mathcal{P}(\mathbb{N}) \to \mathcal{P}(\mathbb{N})$ を考えよう。すると、$\mu F = \{0\}$ かつ $\nu F = \mathbb{N}$ である。さらに、任意の $n \in \nu F \setminus \mu F$、すなわち任意の $n > 0$ について、$support(n) = \{n+1\}$ であり、無限列が生成される。

[†9] (訳注:原著正誤表より) μS が有限型だけしか関係させないというのはあまり正しくない。(無限型を含む) すべての型は Top の部分型であるためである。そのため、より複雑な議論が必要になる。

解答 21.5.13.　まず、部分正当性を考えよう。それぞれの部分の証明はアルゴリズムの実行の再帰的な構造についての帰納法で示せる。

(1) lfp の定義から、$lfp(X)$ が $true$ を返すことのできる場合が二通りあることが、容易に確認できる。$X = \emptyset$ であるために $lfp(X) = true$ である場合、$X \subseteq \mu F$ は自明に成り立つ。一方、$lfp(support(X)) = true$ であるために $lfp(X) = true$ である場合、帰納法の仮定により $support(X) \subseteq \mu F$ である。これに対して補題 21.5.8 を用いることで $X \subseteq \mu F$ が得られる。
(2) $support(X)\uparrow$ であるために $lfp(X) = false$ である場合、補題 21.5.8 によって $X \nsubseteq \mu F$ である。そうでない場合、$lfp(support(X)) = false$ であるため $lfp(X) = false$ であり、帰納法の仮定より $support(X) \nsubseteq \mu F$ である。補題 21.5.8 を用いて、$X \nsubseteq \mu F$ が得られる。

次に、lfp がすべての有限の入力に対して停止することが保障されるような、生成関数 F を特徴付けたい。これを行うためには幾つかの新たな用語を導入すると便利である。有限状態の生成関数 $F \in \mathcal{P}(\mathcal{U}) \to \mathcal{P}(\mathcal{U})$ が与えられたとき、部分関数 $height_F \in \mathcal{U} \rightharpoonup \mathbb{N}$（単に $height$ とも書く）を、以下の条件を満たす最小の部分関数であるとする[†10]。

$$height(x) = \begin{cases} 0 & support(x) = \emptyset \text{ の場合} \\ 0 & support(x)\uparrow \text{ の場合} \\ 1 + max\{height(y) \mid y \in support(x)\} & support(x) \neq \emptyset \text{ の場合} \end{cases}$$

（x が循環に含まれるか、循環に含まれる要素に依存している場合に、$height(x)$ が未定義であることに注意されたい。）生成関数 F は、$height_F$ が全域関数であるときに、**有限の高さ**であるという。$y \in support(x)$ であり、$height(x)$ と $height(y)$ が共に定義されているのなら、$height(y) < height(x)$ である。これは容易に確認できる。

ここで、F が有限状態であり、有限の高さであるなら、任意の有限の入力集合 $X \subseteq \mathcal{U}$ に対して、$lfp(X)$ は停止する。これを確認するためには、F は有限状態であるため、元の呼び出し $lfp(X)$ からの任意の再帰呼び出し $lfp(Y)$ について、Y は有限であるということを観察せよ。F は有限の高さであるので、$h(Y) = max\{height(y) \mid y \in Y\}$ は正しく定義される。$h(Y)$ は再帰呼び出しのたびに減少し、かつ常に非負であるため、lfp の停止尺度として用いることができる。

解答 21.8.5.　S_d の定義は、最後の節が $\mathsf{T} \neq \mu\mathsf{X.T_1}$ と $\mathsf{T} \neq \mathsf{Top}$ という条件を含まない、という点を除いて S_m と同じである。S_d が可逆でないことを確認するには、集合 $G_{(\mu\mathsf{X.Top},\mu\mathsf{Y.Top})}$ が二つの極小な生成集合、すなわち $\{(\mathsf{Top},\mu\mathsf{Y.Top})\}$ および $\{(\mu\mathsf{X.Top},\mathsf{Top})\}$ を含んでいることを観察せよ（S_m 関数の場合の集合の中身と比較せよ）。

S_d と S_m は最後の節を除いて等しく、S_m の最後の節は、S_d の最後の節の制限になっているため、包含関係 $\nu S_m \subseteq \nu S_d$ は明らかである。逆向きの包含関係 $\nu S_d \subseteq \nu S_m$ は、余帰納法の原理と、νS_d が S_m について整合的であることを示す以下の補題とを用いることで証明できる。

補題 A.17.　任意の二つの μ 型 S と T とについて、$(\mathsf{S},\mathsf{T}) \in \nu S_d$ ならば、$(\mathsf{S},\mathsf{T}) \in S_m(\nu S_d)$　□

［証明の概略］　$k = \mu\text{-}height(\mathsf{S})$ についての帰納法による。この帰納法によって、「任意の $(\mathsf{S},\mathsf{T}) \in \nu S_d$ の導出は、同じ事実の導出であって、同時に $(\mathsf{S},\mathsf{T}) \in \nu S_m$ の導出でもあるものへと変形できる」という非形式的なアイディアを検証できる。左 μ 畳み込み規則への制限によって、変形結果の導出は「任意の μ 畳み込み規則の適用列は、左 μ 畳み込みの並びで始まり、その後に右 μ 畳み込みの並びが続く」ことが強制される。　□

[†10] $height$ の定義のこの表現は、部分関数を関係として表現したものの上の単調な関数の最小不動点として容易に言い換えられる、ということを確認せよ。

解答 21.9.2.

$$\frac{}{\mathsf{T} \sqsubseteq \mathsf{T}} \qquad \frac{\mathsf{S} \sqsubseteq \mathsf{T}_1}{\mathsf{S} \sqsubseteq \mathsf{T}_1 \times \mathsf{T}_2} \qquad \frac{\mathsf{S} \sqsubseteq \mathsf{T}_2}{\mathsf{S} \sqsubseteq \mathsf{T}_1 \times \mathsf{T}_2}$$

$$\frac{\mathsf{S} \sqsubseteq \mathsf{T}_1}{\mathsf{S} \sqsubseteq \mathsf{T}_1 \to \mathsf{T}_2} \qquad \frac{\mathsf{S} \sqsubseteq \mathsf{T}_2}{\mathsf{S} \sqsubseteq \mathsf{T}_1 \to \mathsf{T}_2} \qquad \frac{\mathsf{S} \sqsubseteq [\mathsf{X} \mapsto \mu\mathsf{X}.\mathsf{T}]\mathsf{T}}{\mathsf{S} \sqsubseteq \mu\mathsf{X}.\mathsf{T}}$$

(肝心な点として、生成関数 *TD* はこの章を通して取り上げてきた生成関数と異なり、可逆ではない、ということに注意されたい。例えば、$\mathsf{B} \sqsubseteq \mathsf{A} \times \mathsf{B} \to \mathsf{B} \times \mathsf{C}$ は二つの集合 $\{\mathsf{B} \sqsubseteq \mathsf{A} \times \mathsf{B}\}$ および $\{\mathsf{B} \sqsubseteq \mathsf{B} \times \mathsf{C}\}$ によって支持されるが、いずれも他方の部分集合にはなっていない。)

解答 21.9.7. *BU* のための規則は、μ 束縛子で始まる型のための以下の規則を除いて、演習 21.9.2 の解答で示した *TD* のための規則と同じである。

$$\frac{\mathsf{S} \preceq \mathsf{T}}{[\mathsf{X} \mapsto \mu\mathsf{X}.\mathsf{T}]\mathsf{S} \preceq \mu\mathsf{X}.\mathsf{T}}$$

解答 21.11.1. 多数存在する。自明な例は、ほぼ任意の T に対する $\mu\mathsf{X}.\mathsf{T}$ と $[\mathsf{X} \mapsto \mu\mathsf{X}.\mathsf{T}]\mathsf{T}$ である。より興味深いものとしては、$\mu\mathsf{X}.\ \mathsf{Nat} \times (\mathsf{Nat} \times \mathsf{X})$ と $\mu\mathsf{X}.\ \mathsf{Nat} \times \mathsf{X}$ が挙げられる。

解答 22.3.9. 主なアルゴリズム的制約生成規則を示す。

$$\frac{\Gamma(\mathsf{x}) = \mathsf{T}}{\Gamma \vdash_F \mathsf{x} : \mathsf{T} \mid_F \{\}} \tag{CT-Var'}$$

$$\frac{\Gamma, \mathsf{x}{:}\mathsf{T}_1 \vdash_F \mathsf{t}_2 : \mathsf{T}_2 \mid_{F'} C \qquad \mathsf{x} \notin dom(\Gamma)}{\Gamma \vdash_F \lambda\mathsf{x}{:}\mathsf{T}_1.\mathsf{t}_2 : \mathsf{T}_1 \to \mathsf{T}_2 \mid_{F'} C} \tag{CT-Abs'}$$

$$\frac{\Gamma \vdash_F \mathsf{t}_1 : \mathsf{T}_1 \mid_{F'} C_1 \qquad \Gamma \vdash_{F'} \mathsf{t}_2 : \mathsf{T}_2 \mid_{F''} C_2 \qquad F'' = \mathsf{X}, F'''}{\Gamma \vdash_F \mathsf{t}_1\ \mathsf{t}_2 : \mathsf{X} \mid_{F'''} C_1 \cup C_2 \cup \{\mathsf{T}_1 = \mathsf{T}_2 \to \mathsf{X}\}} \tag{CT-App'}$$

他の規則も同様である。元の規則とアルゴリズム的規則の等価性は次のような言明で説明できる。

(1) [健全性] $\Gamma \vdash_F \mathsf{t} : \mathsf{T} \mid_{F'} C$ かつ Γ か t に現れる変数が F に出現しないならば、$\Gamma \vdash \mathsf{t} : \mathsf{T} \mid_{F \setminus F'} C$

(2) [完全性] $\Gamma \vdash \mathsf{t} : \mathsf{T} \mid_{\mathcal{X}} C$ ならば、$\mathcal{X}$ の名前の並べ替え F が存在し、$\Gamma \vdash_F \mathsf{t} : \mathsf{T} \mid_{\varnothing} C$

いずれも導出に関する単純な帰納法で証明される。(1) の適用の場合には、以下の補題が有用である。

> Γ や t に現れる型変数が F に含まれず、さらに $\Gamma \vdash_F \mathsf{t} : \mathsf{T} \mid_{F'} C$ ならば、T や C に現れる型変数はすべて、Γ、t、$F \setminus F'$ のいずれかに現れる。

(2) の適用の場合には、以下の補題を用いる。

> $\Gamma \vdash_F \mathsf{t} : \mathsf{T} \mid_{F'} C$ ならば、$\Gamma \vdash_{F,G} \mathsf{t} : \mathsf{T} \mid_{F',G} C$。ただし G はフレッシュな変数名からなる任意の列である。

解答 22.3.10.　制約集合を型の二つ組のリストとして表現すると、制約生成アルゴリズムは演習 22.3.9 の解答の推論規則をそのまま書き写したものとなる。

```
let rec recon ctx nextuvar t = match t with
      TmVar(fi,i,_) →
        let tyT = getTypeFromContext fi ctx i in
        (tyT, nextuvar, [])
    | TmAbs(fi, x, tyT1, t2) →
        let ctx' = addbinding ctx x (VarBind(tyT1)) in
        let (tyT2,nextuvar2,constr2) = recon ctx' nextuvar t2 in
        (TyArr(tyT1, tyT2), nextuvar2, constr2)
    | TmApp(fi,t1,t2) →
        let (tyT1,nextuvar1,constr1) = recon ctx nextuvar t1 in
        let (tyT2,nextuvar2,constr2) = recon ctx nextuvar1 t2 in
        let NextUVar(tyX,nextuvar') = nextuvar2() in
        let newconstr = [(tyT1,TyArr(tyT2,TyId(tyX)))] in
        ((TyId(tyX)), nextuvar',
         List.concat [newconstr; constr1; constr2])
    | TmZero(fi) → (TyNat, nextuvar, [])
    | TmSucc(fi,t1) →
        let (tyT1,nextuvar1,constr1) = recon ctx nextuvar t1 in
        (TyNat, nextuvar1, (tyT1,TyNat)::constr1)
    | TmPred(fi,t1) →
        let (tyT1,nextuvar1,constr1) = recon ctx nextuvar t1 in
        (TyNat, nextuvar1, (tyT1,TyNat)::constr1)
    | TmIsZero(fi,t1) →
        let (tyT1,nextuvar1,constr1) = recon ctx nextuvar t1 in
        (TyBool, nextuvar1, (tyT1,TyNat)::constr1)
    | TmTrue(fi) → (TyBool, nextuvar, [])
    | TmFalse(fi) → (TyBool, nextuvar, [])
    | TmIf(fi,t1,t2,t3) →
        let (tyT1,nextuvar1,constr1) = recon ctx nextuvar t1 in
        let (tyT2,nextuvar2,constr2) = recon ctx nextuvar1 t2 in
        let (tyT3,nextuvar3,constr3) = recon ctx nextuvar2 t3 in
        let newconstr = [(tyT1,TyBool); (tyT2,tyT3)] in
        (tyT3, nextuvar3,
         List.concat [newconstr; constr1; constr2; constr3])
```

解答 22.3.11.　`fix` 式の制約生成規則は、図 11-12 の型付け規則 T-FIX から単純に導出できる。

$$\frac{\Gamma \vdash \mathtt{t}_1 : \mathtt{T}_1 \ |_{\mathcal{X}_1} C_1 \qquad \mathtt{X}\text{ は }\mathcal{X}_1, \Gamma, \mathtt{t}_1\text{ のいずれにも現れない}}{\Gamma \vdash \mathtt{fix}\ \mathtt{t}_1 : \mathtt{X} \ |_{\mathcal{X}_1 \cup \{\mathtt{X}\}} C_1 \{\mathtt{T}_1 = \mathtt{X}\to\mathtt{X}\}} \quad \text{(CT-FIX)}$$

この規則は、$\mathtt{t}_1$ の型（$\mathtt{T}_1$ と呼ぶ）を再構築し、$\mathtt{T}_1$ があるフレッシュな X に対して X→X という形を持つようにし、$\mathtt{fix}\ \mathtt{t}_1$ の型として X を返す。

`letrec` 式の制約生成規則は、`letrec` を派生形式とする定義と併せて、`fix` 式の規則から導出できる。

解答 22.4.3.

{X = Nat, Y = X→X}	[X ↦ Nat, Y ↦ Nat→Nat]
{Nat→Nat = X→Y}	[X ↦ Nat, Y ↦ Nat]
{X→Y = Y→Z, Z = U→W}	[X ↦ U→W, Y ↦ U→W, Z ↦ U→W]
{Nat = Nat→Y}	単一化不可能
{Y = Nat→Y}	単一化不可能
{ }	[ ]

解答 22.4.6.　この演習で必要になる主なデータ構造は、代入の表現である。選択肢は幾つもあるが、演習 22.3.10 の constr データ型を再利用するのが単純な方法の一つである。つまり、代入を、左辺が常に単一化変数であるような制約集合とみなす。一つの型変数に型を代入する関数 substinty を定義する。

```
let substinty tyX tyT tyS =
  let rec f tyS = match tyS with
      TyArr(tyS1,tyS2) → TyArr(f tyS1, f tyS2)
    | TyNat → TyNat
    | TyBool → TyBool
    | TyId(s) → if s=tyX then tyT else TyId(s)
  in f tyS
```

すると、代入全体を型に適用する関数は以下のように定義できる。

```
let applysubst constr tyT =
  List.fold_left
    (fun tyS (TyId(tyX),tyC2) → substinty tyX tyC2 tyS)
    tyT (List.rev constr)
```

制約集合に含まれるすべての型に対して代入を適用する単一化関数も必要である。

```
let substinconstr tyX tyT constr =
  List.map
    (fun (tyS1,tyS2) →
       (substinty tyX tyT tyS1, substinty tyX tyT tyS2))
    constr
```

他に重要なのは、依存関係の循環を検出する「出現検査」である。

```
let occursin tyX tyT =
  let rec o tyT = match tyT with
      TyArr(tyT1,tyT2) → o tyT1 || o tyT2
    | TyNat → false
    | TyBool → false
    | TyId(s) → (s=tyX)
  in o tyT
```

以上を用いると、単一化関数は図 22-2 で与えた擬似コードをそのまま書き写したものとなる。いつものように、追加の引数としてファイル中の位置と文字列を受け取る。これらは単一化に失敗したときにエラーメッセージを表示するために用いられる。

```
let unify fi ctx msg constr =
  let rec u constr = match constr with
      [] → []
    | (tyS,TyId(tyX)) :: rest →
        if tyS = TyId(tyX) then u rest
        else if occursin tyX tyS then
          error fi (msg ^ ": circular constraints")
        else
          List.append (u (substinconstr tyX tyS rest))
                      [(TyId(tyX),tyS)]
    | (TyId(tyX),tyT) :: rest →
        if tyT = TyId(tyX) then u rest
        else if occursin tyX tyT then
          error fi (msg ^ ": circular constraints")
        else
          List.append (u (substinconstr tyX tyT rest))
                      [(TyId(tyX),tyT)]
    | (TyNat,TyNat) :: rest → u rest
    | (TyBool,TyBool) :: rest → u rest
    | (TyArr(tyS1,tyS2),TyArr(tyT1,tyT2)) :: rest →
        u ((tyS1,tyT1) :: (tyS2,tyT2) :: rest)
    | (tyS,tyT)::rest →
        error fi "Unsolvable constraints"
  in
    u constr
```

この単一化関数は教育目的なので、わかりやすいエラーメッセージを表示するための努力はあまり行っていない。しかし実用的には、型エラーを「説明」することは、型再構築を備える言語の製品レベルのコンパイラを開発する上で最も難しい部分の一つとなりうる。Wand [486] を参照せよ。

解答 22.5.6.　型再構築アルゴリズムを拡張してレコードを扱えるようにするのは、単純ではないが可能である。主に難しいのは、レコードの射影に対してどのような制約を生成するべきかが明らかでないことである。最初に思いつく素朴な方法は以下のようなものだろう。

$$\frac{\Gamma \vdash \mathtt{t}_1 : \mathtt{T} \ \ |_{\mathcal{X}}\ C}{\Gamma \vdash \mathtt{t}_1.\mathtt{l}_i : \mathtt{X} \ \ |_{\mathcal{X}\cup\{\mathtt{X}\}}\ C \cup \{\mathtt{T} = \{\mathtt{l}_i{:}\mathtt{X}\}\}}$$

しかしこれは満足できるものではない。この規則は事実上、フィールド $\mathtt{l}_i$ を射影できるのはフィールド $\mathtt{l}_i$ ***だけ***を持っているレコードだけである、といっている。

美しい解が Wand [487] によって提案され、Wand [488; 490] や Remy [399; 400] などがそれをさらに発展させた。**列変数**と呼ばれる新たな種類の変数を導入する。この変数は型の上を動くのではなく、フィールドラベルとそれに関連付けられた型の「列」の上を動く。列変数を用いると、フィールド射影の制約生成規則は以下のように書ける。

$$\frac{\Gamma \vdash \mathtt{t}_1 : \mathtt{T} \ \ |_{\mathcal{X}}\ C}{\Gamma \vdash \mathtt{t}_1.\mathtt{l}_i : \mathtt{X} \ \ |_{\mathcal{X}\cup\{\mathtt{X},\sigma,\rho\}}\ C \cup \{\mathtt{T} = \{\rho\},\ \rho = \mathtt{l}_i{:}\mathtt{X} \oplus \sigma\}} \qquad \text{(CT-Proj)}$$

ただし σ と ρ は列変数であり、$\oplus$ は二つの列を結合する演算子である（それぞれのフィールドは異なるものと仮定する）。つまり、ρ は σ が含むフィールドすべてとフィールド $\mathtt{l}_i{:}\mathtt{X}$ を含むものとし、項 $\mathtt{t}_1$ が、ρ が含むすべてのフィールドを持つレコード型であるならば、項 $\mathtt{t}_1.\mathtt{l}_i$ は型 X を持つ。

この改良されたアルゴリズムが生成する制約は、元の再構築アルゴリズムで用いていた、単一化

変数を含む型の等式の単純な集合に比べると、より複雑なものになっている。これは、新しい制約集合は結合的で可換な演算子 ⊕ を含むためである。このような制約集合の解を見つけるには、一種の単純な**等式単一化**が必要である。

解答 23.4.3. 以下に、append を補助的に用いる標準的な解答を示す。

```
append = λX. (fix (λapp:List X → List X → List X.
                λl1:List X. λl2:List X.
                  if isnil [X] l1 then l2
                  else cons [X] (head [X] l1)
                                (app (tail [X] l1) l2)));
▸ append : ∀X. List X → List X → List X

reverse =
  λX.
   (fix (λrev:List X → List X.
          λl: List X.
           if isnil [X] l
           then nil [X]
           else append [X] (rev (tail [X] l))
                           (cons [X] (head [X] l) (nil [X]))));
▸ reverse : ∀X. List X → List X
```

解答 23.4.5.

```
and = λb:CBool. λc:CBool.
        λX. λt:X. λf:X. b [X] (c [X] t f) f;
```

解答 23.4.6.

```
iszro = λn:CNat. n [CBool] (λb:CBool. fls) tru;
```

解答 23.4.8.

```
pairNat = λn1:CNat. λn2:CNat.
            λX. λf:CNat→CNat→X. f n1 n2;
fstNat = λp:PairNat. p [CNat] (λn1:CNat. λn2:CNat. n1);
sndNat = λp:PairNat. p [CNat] (λn1:CNat. λn2:CNat. n2);
```

解答 23.4.9.

```
zz = pairNat c0 c0;
f = λp:PairNat. pairNat (sndNat p) (cplus c1 (sndNat p));
prd = λm:CNat. fstNat (m [PairNat] f zz);
```

解答 23.4.10.

```
vpred = λn:CNat. λX. λs:X→X.
            λz:X.
              (n [(X→X)→X]
                 (λp:(X→X)→X. λq:(X→X). q (p s))
                 (λx:X→X. z))
                (λx:X. x);
▸ vpred : CNat → CNat
```

この例を教えてくれた Michael Levin に感謝する。

解答 23.4.11.

```
head = λX. λdefault:X. λl:List X.
          l [X] (λhd:X. λtl:X. hd) default;
```

解答 23.4.12. この演習で最もトリッキーなのは挿入関数である。その解は、与えられたリスト l を、二つの新しいリストを構築する関数に適用することでうまく動作する。その二つとは、一つは元のリストと同じもの、もう一つは e を含むものである。この関数は、l の各要素 hd（これは右から左へと辿られる）について、引数として hd と、さらに hd より右の要素に対して構築した二つのリストの組を受け取る。それから e と hd を比較し、新しいリストの組を構築して返す。もし e が hd より小さいか等しければ、e は返り値の二つ目のリストの先頭に位置するべきである。したがって、渡された一つ目のリスト（まだ e を含んでいないほう）の先頭に e を追加することで、返り値の二つ目のリストを構築する。一方、e が hd より大きい場合は、e は二つ目のリスト中のどこかに位置するべきなので、既に構築して渡された二つ目のリストに hd を追加することで、返り値の二つ目のリストを構築する。

```
insert =
  λX. λleq:X→X→Bool. λl:List X. λe:X.
    let res =
      l [Pair (List X) (List X)]
        (λhd:X. λacc: Pair (List X) (List X).
           let rest = fst [List X] [List X] acc in
           let newrest = cons [X] hd rest in
           let restwithe = snd [List X] [List X] acc in
           let newrestwithe =
              if leq e hd
                then cons [X] e (cons [X] hd rest)
                else cons [X] hd restwithe in
           pair [List X] [List X] newrest newrestwithe)
        (pair [List X] [List X] (nil [X]) (cons [X] e (nil [X])))
    in snd [List X] [List X] res;
▸ insert : ∀X. (X→X→Bool) → List X → X → List X
```

次に数の比較関数が必要である。プリミティブの数を用いているから、比較を書くには fix を用いる必要がある。（Nat でなく CNat を用いていたら、fix を使わないで済む。）

```
leqnat =
  fix (λf:Nat→Nat→Bool. λm:Nat. λn:Nat.
         if iszero m then true
         else if iszero n then false
         else f (pred m) (pred n));

▸ leqnat : Nat → Nat → Bool
```

最後に、リストの各要素を新しいリストに順次挿入していくことでソート関数を作る。

```
sort = λX. λleq:X→X→Bool. λl:List X.
          l [List X]
            (λhd:X. λrest:List X. insert [X] leq rest hd)
            (nil [X]);
▸ sort : ∀X. (X→X→Bool) → List X → List X
```

sortが正しく動くことを確認するため、ソートされていないリストを作る。

```
l = cons [Nat] 9
      (cons [Nat] 2 (cons [Nat] 6 (cons [Nat] 4 (nil [Nat]))));
```

これをソートし、

```
l = sort [Nat] leqnat l;
```

その要素を読み出す。

```
  nth =
    λX. λdefault:X.
      fix (λf:List X→Nat→X. λl:List X. λn:Nat.
             if iszero n
               then head [X] default l
               else f (tail [X] l) (pred n));
▸ nth : ∀X. X → List X → Nat → X

  nth [Nat] 0 l 0;
▸ 2 : Nat

  nth [Nat] 0 l 1;
▸ 4 : Nat

  nth [Nat] 0 l 2;
▸ 6 : Nat

  nth [Nat] 0 l 3;
▸ 9 : Nat

  nth [Nat] 0 l 4;
▸ 0 : Nat
```

このように、正しく型付けされたソート関数がSystem Fで実装できることを実演したのは、Reynolds [407] による**偉業**である。ただしReynoldsのアルゴリズムはここで示したものとは若干異なる。

解答 23.5.1. 証明の構造は定理 9.3.9（81 ページ）とほぼ完全に同じである。型適用の規則 E-TAPPTABS に対して、もう一つ代入に関する補題が必要である。これは補題 9.3.8（80 ページ）に相当する。

$\Gamma, X, \Delta \vdash t : T$ ならば、$\Gamma, [X \mapsto S]\Delta \vdash [X \mapsto S]t : [X \mapsto S]T$

Δ という余分な文脈は、十分強力な帰納法の仮定を得るために必要である。これがないと、T-ABS の場合で証明に詰まる。

解答 23.5.2. これも証明の構造は $\lambda_{\rightarrow}$ の進行（定理 9.3.5）の証明と非常に似ている。標準形補題（補題 9.3.4）には場合が一つ増える。

v が型 $\forall X.T_{12}$ の値ならば、$v = \lambda X.t_{12}$

これは本体の証明の、型適用の場合に用いられる。

解答 23.6.3. どの問題も、比較的単純な帰納法や計算による。ただし最後の問題だけは別で、他の解答をつなぎ合わせて最終的な結論を得るために、洞察が少し必要である。この議論の構成は Pawel Urzyczyn が提案した。

(1) t に関する単純な帰納法。型付けの逆転補題を用いる。

(2) 外側の型抽象と適用の数に関する帰納法で以下を証明する。

ある $\overline{\mathtt{Y}}$、$\overline{\mathtt{B}}$、r に対して、t が $\lambda\overline{\mathtt{Y}}.$ (r $[\overline{\mathtt{B}}]$) という形を持ち（r は露出している必要はない）、*erase*(t) = m かつ $\Gamma \vdash$ t : T ならば、s = $\lambda\overline{\mathtt{X}}.$ (u $[\overline{\mathtt{A}}]$) という形の型 s が存在して、*erase*(s) = m かつ $\Gamma \vdash$ s : T。さらに、u は露出している。

基底ケースでは、外側に型抽象や適用は存在しない。つまり r 自身が露出しているので、証明が完了する。

帰納ケースでは、r の外側の構築子は型抽象か型適用である。もし型適用であれば（$\mathtt{r_1}$ [R] とする）、R を列 $\overline{\mathtt{B}}$ に追加し、帰納法の仮定を適用する。もし型抽象であれば（λZ. $\mathtt{r_1}$ とする）、さらに二つの場合に分けて考える。

(a) 適用の列 $\overline{\mathtt{B}}$ が空であれば、Z を抽象の列 $\overline{\mathtt{Y}}$ に追加し、帰納法の仮定を適用できる。

(b) $\overline{\mathtt{B}}$ が空でなければ、t を以下のように書ける。

$$\mathtt{t} = \lambda\overline{\mathtt{Y}}.\ ((\lambda\mathtt{Z}.\ \mathtt{r_1})\ [\mathtt{B_0}]\ [\overline{\mathtt{B}}'])$$

ただし $\overline{\mathtt{B}} = \mathtt{B_0}\overline{\mathtt{B}}'$ である。ここでこの項は E-TappTabs の簡約基を持つ。この簡約基を簡約すると、以下の項になる。

$$\mathtt{t}' = \lambda\overline{\mathtt{Y}}.\ ([\mathtt{B_0} \mapsto \mathtt{Z}]\mathtt{r_1}\ [\overline{\mathtt{B}}'])$$

ただし $[\mathtt{B_0} \mapsto \mathtt{Z}]\mathtt{r_1}$ が持つ外側の型抽象や適用の数は r より少ない。さらに、主部簡約定理から、t′ が t と同じ型を持つことがわかる。ここで帰納法の仮定を適用すれば所望の結果が得られる。

(3) 逆転補題から直ちに証明できる。

(4) (2) と (3) を用いて単純に計算する。

(5) (2) と逆転補題から直ちに得られる。

(6) $\mathtt{T_1}$ のサイズに関する帰納法による。基底ケースでは、$\mathtt{T_1}$ は変数であり、この変数は $\overline{\mathtt{X}}_1\overline{\mathtt{X}}_2$ に含まれるものでなければならない。そうでないとしたら、

$$[\overline{\mathtt{X}}_1\overline{\mathtt{X}}_2 \mapsto \overline{\mathtt{A}}](\forall\overline{\mathtt{Y}}.\mathtt{T_1}) \quad = \quad \forall\overline{\mathtt{Y}}.\mathtt{W} \quad = \quad \forall\overline{\mathtt{Z}}.(\forall\overline{\mathtt{Y}}.\mathtt{W})\to([\overline{\mathtt{X}}_1 \mapsto \overline{\mathtt{B}}]\mathtt{T_2})$$

となり、こういう場合はありえないからである（左側には矢印がなく、右側には少なくとも一つの矢印がある）。他の場合は帰納法の仮定から直ちに得られる。

(7) 背理法を用いる。omega が型付け可能であると仮定する。このとき、(1) と (3) から、ある露出している項 o = s u が存在し、

erase(s) = λx. x x　　*erase*(u) = λy. y y
$\Gamma \vdash$ s : U→V　　$\Gamma \vdash$ u : U

である。(2) から、項 s′ = $\lambda\overline{\mathtt{R}}.$ ($\mathtt{s_0}$ $[\overline{\mathtt{E}}]$) と u′ = $\lambda\overline{\mathtt{V}}.$ ($\mathtt{u_0}$ $[\overline{\mathtt{F}}]$) が存在する。ただし $\mathtt{s_0}$ と $\mathtt{u_0}$ は露出していて、

erase($\mathtt{s_0}$) = λx. x x　*erase*($\mathtt{u_0}$) = λy. y y
$\Gamma \vdash$ s′ : U→V　　$\Gamma \vdash$ u′ : U

である。s′ は矢印型を持つから、$\overline{\mathtt{R}}$ は空でなければならない。同様に、$\mathtt{s_0}$ と $\mathtt{u_0}$ は露出して

いるから、これらは共に抽象で始まらなければならない。よって $\overline{E}$ と $\overline{F}$ も空である。したがって、

$$\begin{aligned} o' &= s'\ u' \\ &= s_0\ (\lambda\overline{V}.\ u_0) \\ &= (\lambda x{:}T_x.\ w)\ (\lambda\overline{V}.\ \lambda y{:}T_y.\ v) \end{aligned}$$

ただし *erase*(w) = x x かつ *erase*(v) = y y である。逆転補題により、$U = T_x$ かつ

$$\Gamma, x{:}T_x \vdash w : W \qquad \Gamma, \overline{V}, y{:}T_y \vdash v : P$$

である。(4) を一つ目に適用すると、以下のいずれかがいえる。

(a) $T_x = \forall\overline{X}.X_i$

(b) $T_x = \forall\overline{X}_1\overline{X}_2.T_1\to T_2$、かつ、ある $\overline{A}$ と $\overline{B}$ が存在して
$[\overline{X}_1\overline{X}_2 \mapsto \overline{A}]T_1 = [\overline{X}_1 \mapsto \overline{B}](\forall\overline{Z}.T_1\to T_2)$

(5) より、T_x は二つ目の形を持たなければならない。よって (6) により、T_1 の最左の葉は $X_i \in \overline{X}_1\overline{X}_2$ である。

ここで、(4) を型付け $\Gamma, \overline{V}, y{:}T_y \vdash v : P$ に適用すると、以下のいずれかがいえる。

(a) $T_y = \forall\overline{Y}.Y_i$

(b) $T_y = \forall\overline{Y}_1\overline{Y}_2.S_1\to S_2$、かつ、ある $\overline{C}$ と $\overline{C}$ が存在して
$[\overline{Y}_1\overline{Y}_2 \mapsto C]S_1 = [\overline{Y}_1 \mapsto D](\forall\overline{Z}'.S_1\to S_2)$

前者の場合、直ちに T_y の最左の葉が $Y_i \in \overline{Y}$ だとわかる。後者の場合、(6) を用いると、やはり T_y の最左の葉が $Y_i \in \overline{Y}_1\overline{Y}_2$ であるとわかる。

しかし o′ の形と逆転補題から、以下がいえる。

$$\begin{aligned} \forall\overline{V}.T_y\to V &= T_x \\ &= \forall\overline{X}_1\overline{X}_2.T_1\to T_2 \end{aligned}$$

よって、特に $T_y = T_1$ である。つまり、T_1 の最左の葉は T_y の最左の葉と同じである。まとめると、$T_x = \forall\overline{X}_1\overline{X}_2.(\forall\overline{Y}.S)\to T_2$ であり、ただし *leftmost-leaf*(S) = $X_i \in \overline{X}_1\overline{X}_2$ かつ *leftmost-leaf*(S) = $Y_i \in \overline{Y}$。しかし変数 $\overline{X}_1\overline{X}_2$ と $\overline{Y}$ は他の場所で束縛されているので、矛盾が導かれた。omega が型付け可能という最初の仮定が偽であったということになる。

解答 23.7.1.

```
let r = λX. ref (λx:X. x) in
(r[Nat] := (λx:Nat. succ x);
 (!(r[Bool])) true);
```

解答 24.1.1. パッケージ p6 は定数 a と関数 f を提供するが、このような型では、a に f を何回か適用してその結果を捨てることしかできない。パッケージ p7 では f を用いて型 X の値を作れるが、そうして作られた値に対して何もできない。p8 では、どちらの要素も使用可能だが、もはや何も隠されていない。いっそ存在型のパッケージ化もやめてしまえばよいかもしれない。

解答 24.2.1.

```
stackADT =
   {*List Nat,
    {new = nil [Nat],
     push = λn:Nat. λs:List Nat. cons [Nat] n s,
     top = λs:List Nat. head [Nat] s,
     pop = λs:List Nat. tail [Nat] s,
     isempty = isnil [Nat]}}
  as {∃Stack, {new: Stack, push: Nat→Stack→Stack, top: Stack→Nat,
               pop: Stack→Stack, isempty: Stack→Bool}};
▸ stackADT : {∃Stack,
              {new:Stack,push:Nat→Stack→Stack,top:Stack→Nat,
               pop:Stack→Stack,isempty:Stack→Bool}}

let {Stack,stack} = stackADT in
stack.top (stack.push 5 (stack.push 3 stack.new));
▸ 5 : Nat
```

解答 24.2.2.

```
counterADT =
   {*Ref Nat,
    {new = λ_:Unit. ref 1,
     get = λr:Ref Nat. !r,
     inc = λr:Ref Nat. r := succ(!r)}}
   as {∃Counter,
       {new: Unit→Counter, get: Counter→Nat, inc: Counter→Unit}};
▸ counterADT : {∃Counter,
                {new:Unit→Counter,get:Counter→Nat,
                 inc:Counter→Unit}}
```

解答 24.2.3.

```
FlipFlop = {∃X, {state:X, methods: {read: X→Bool, toggle: X→X,
                                    reset: X→X}}};
f = {*Counter,
     {state = zeroCounter,
      methods = {read = λs:Counter. iseven (sendget s),
                 toggle = λs:Counter. sendinc s,
                 reset = λs:Counter. zeroCounter}}}
    as FlipFlop;
▸ f : FlipFlop
```

解答 24.2.4.

```
c = {*Ref Nat,
     {state = ref 5,
      methods = {get = λx:Ref Nat. !x,
                 inc = λx:Ref Nat. (x := succ(!x); x)}}}
    as Counter;
```

解答 24.2.5. この型なら union メソッドのある集合オブジェクトを実装することはできるが、使用することができない。このようなオブジェクトの union メソッドを呼び出すには、全く同じ表現型 X を持つ二つの値を渡す必要がある。しかし、これらの値として二つの異なる集合オブジェクト

を用いることはできない。なぜなら、二つのオブジェクトの状態を得るためにそれぞれのオブジェクトを開く必要があるが、そうすると二つの相異なる型変数を束縛することになるからである。つまり、二つ目の集合の状態を一つ目の集合の `union` 演算に渡すことができないのだ（これは、単に型検査器が頑固だという話ではない。一つの集合の具象表現をもう一つの `union` 演算に渡すことが健全でないことはすぐわかる。なぜなら、一般には、二つ目の表現は一つ目とは任意に異なる可能性があるからだ)。つまりこの `NatSet` 型の集合が和集合を計算できる相手は、自分自身だけということなのである。

解答 24.3.2. 最低でも、この翻訳を行っても存在型の型付け規則と計算規則が保存されることを示す必要がある。すなわち、これらの翻訳全部を行う関数を $[\![-]\!]$ と書くとすれば、$\Gamma \vdash \mathtt{t} : \mathtt{T}$ ならば $[\![\Gamma]\!] \vdash [\![\mathtt{t}]\!] : [\![\mathtt{T}]\!]$ であることと、$\mathtt{t} \longrightarrow^* \mathtt{t}'$ ならば $[\![\mathtt{t}]\!] \longrightarrow^* [\![\mathtt{t}']\!]$ であることである。これらの性質が成り立つことはすぐに確認できる。さらに、逆が真であることも証明したいかもしれない。すなわち、存在型を持つ言語において正しく型付けされて**いない**項は、この翻訳を行っても正しく型付けされていない項になること、また、行き詰まり状態の項は行き詰まり状態の項になることである。しかし残念ながら、これらは成り立たない。例えば、正しく型付けされていない（しかも行き詰まり状態である）項 `({*Nat, 0} as {∃X.X}) [Bool]` を翻訳すると、正しく型付けされた（しかも行き詰まり状態でない）項になる。

解答 24.3.3. このことをきちんと書き下したものは見たことがない。これは可能であるはずだが、その変形は局所的な糖衣構文では済まないだろう。つまり、プログラム全体に対して一斉に変形を適用する必要がある。

解答 25.2.1. 型 T の、打ち切り値 c 以上の d 個シフトを $\uparrow^d_c(\mathtt{T})$ と書き、次のように定義する。

$$
\begin{array}{lcl}
\uparrow^d_c(\mathtt{k}) & = & \begin{cases} \mathtt{k} & k < c \text{ の場合} \\ \mathtt{k}+d & k \geq c \text{ の場合} \end{cases} \\
\uparrow^d_c(\mathtt{T_1}\to\mathtt{T_2}) & = & \uparrow^d_c(\mathtt{T_1})\to \uparrow^d_c(\mathtt{T_2}) \\
\uparrow^d_c(\forall.\mathtt{T_1}) & = & \forall.\uparrow^d_{c+1}(\mathtt{t_1}) \\
\uparrow^d_c(\{\exists,\mathtt{T_1}\}) & = & \{\exists,\uparrow^d_{c+1}(\mathtt{T_1})\}
\end{array}
$$

$\uparrow^d(\mathtt{T})$ は、型 T の中のすべての変数を d 個シフトする関数、つまり $\uparrow^d_0(\mathtt{T})$ を表すとする。

解答 25.4.1. 型変数 X のスペースを作るためである。$\mathtt{v_{12}}$ を $\mathtt{t_2}$ に代入した結果は $\Gamma,\mathtt{X}$ という形の文脈において正しいスコープを持つが、元の $\mathtt{v_{12}}$ は単に Γ に対して定義されている。

解答 26.2.3. フルの $F_{<:}$ 規則が必要になる例としては、Abadi, Cardelli, and Viswanathan [9] のオブジェクト表現が挙げられる。Abadi and Cardelli [4] にも記載されている。

解答 26.3.4.

```
   spluszz = λn:SZero. λm:SZero.
                λX. λS<:X. λZ<:X. λs:X→S. λz:Z.
                  n [X] [S] [Z] s (m [X] [S] [Z] s z);
   spluspn = λn:SPos. λm:SNat.
                λX. λS<:X. λZ<:X. λs:X→S. λz:Z.
                  n [X] [S] [X] s (m [X] [S] [Z] s z);
 ▸ spluspn : SPos → SNat → SPos
```

解答 26.3.5.

```
  SBool  = ∀X. ∀T<:X. ∀F<:X. T→F→X;
  STrue  = ∀X. ∀T<:X. ∀F<:X. T→F→T;
  SFalse = ∀X. ∀T<:X. ∀F<:X. T→F→F;
  tru = λX. λT<:X. λF<:X. λt:T. λf:F. t;
▸ tru : STrue

  fls = λX. λT<:X. λF<:X. λt:T. λf:F. f;
▸ fls : SFalse

  notft = λb:SFalse. λX. λT<:X. λF<:X. λt:T. λf:F. b[X][F][T] f t;
▸ notft : SFalse → STrue

  nottf = λb:STrue. λX. λT<:X. λF<:X. λt:T. λf:F. b[X][F][T] f t;
▸ nottf : STrue → SFalse
```

解答 26.4.3. (1) と (2) の抽象と型抽象のケース。また、(3) と (4) の量化子のケース。

解答 26.4.5. (1) では部分型付け導出に関する帰納法を行う。どの場合も直ちに証明できる（S-Refl、S-Top）か、帰納法の仮定を単純に適用するだけ（S-Trans、S-Arrow、S-All）である。ただし S-TVar だけは別で、興味深いケースである。$\Gamma, X<:Q, \Delta \vdash S <: T$ の導出の最後の規則が S-TVar のインスタンスであるとする。つまり S はある変数 Y で、T はこの文脈における Y の上界である。検討すべき可能性が二つある。X と Y が異なる変数である場合、仮定 $Y<:T$ は文脈 $\Gamma, X<:P, \Delta$ の中にもあるので、結論は直ちに得られる。一方 $X = Y$ の場合、$T = Q$ であり、議論を完結するためには、$\Gamma, X<:P, \Delta \vdash X <: Q$ を証明する必要がある。S-TVar より、$\Gamma, X<:P, \Delta \vdash X <: P$ とわかる。さらに、仮定より $\Gamma \vdash P <: Q$。よって弱化（補題 26.4.2）により、$\Gamma, X<:P, \Delta \vdash P <: Q$。これら二つの導出を S-Trans でつなげると、所望の結論が得られる。

(2) は型付け導出に関する定型的な帰納法による。型適用のケースの部分型付けの前提には (1) を用いる。

解答 26.4.11. どの証明も部分型付け導出に関する単純な帰納法による。(1) だけを示す。導出の最後の規則に基づく場合分けを行う。S-Refl と S-Top の場合は直ちに証明できる。S-TVar は起きえない（S-TVar の結論の左辺は変数しかありえない。つまり矢印にはなりえない）。同様に S-All も起きえない。最後の規則が S-Arrow のインスタンスの場合、その部分導出が所望の結論である。残る場合として、最後の規則が S-Trans のインスタンスとする。つまりある U について $\Gamma \vdash S_1 \to S_2 <: U$ かつ $\Gamma \vdash U <: T$ とする。帰納法の仮定により、U は Top である（この場合、演習の (4) により T も Top であり、証明完了）か、または U は $U_1 \to U_2$ という形を持ち、$\Gamma \vdash U_1 <: S_1$ かつ $\Gamma \vdash S_2 <: U_2$ である。後者の場合、元の S-Trans の二つ目の部分導出にも帰納法の仮定を適用すると、$T = Top$ である（証明完了）か、または T は $T_1 \to T_2$ という形を持ち、$\Gamma \vdash T_1 <: U_1$ かつ $\Gamma \vdash U_2 <: T_2$ とわかる。推移律を二回用いると、$\Gamma \vdash T_1 <: S_1$ かつ $\Gamma \vdash S_2 <: T_2$ がわかる。ここから、S-Arrow によって所望の結論が得られる。

解答 26.5.1.

$$\frac{\Gamma \vdash S_1 <: T_1 \qquad \Gamma, X<:S_1 \vdash S_2 <: T_2}{\Gamma \vdash \{\exists X<:S_1, S_2\} <: \{\exists X<:T_1, T_2\}} \qquad \text{(S-Some)}$$

解答 26.5.2. 部分型付けがなければ、四つしかない。

```
{*Nat, {a=5,b=7}} as {∃X, {a:Nat,b:Nat}};
{*Nat, {a=5,b=7}} as {∃X, {a:X,b:Nat}};
{*Nat, {a=5,b=7}} as {∃X, {a:Nat,b:X}};
{*Nat, {a=5,b=7}} as {∃X, {a:X,b:X}};
```

部分型付けと有界量化があれば一気に増える。以下に例を挙げる。

```
{*Nat, {a=5,b=7}} as {∃X, {a:Nat}};
{*Nat, {a=5,b=7}} as {∃X, {b:X}};
{*Nat, {a=5,b=7}} as {∃X, {a:Top,b:X}};
{*Nat, {a=5,b=7}} as {∃X, Top};
{*Nat, {a=5,b=7}} as {∃X<:Nat, {a:X,b:X}};
{*Nat, {a=5,b=7}} as {∃X<:Nat, {a:Top,b:X}};
```

解答 26.5.3. これを達成する一つの方法は、リセットカウンタ ADT をカウンタ ADT の中に入れることである。

```
counterADT =
   {*Nat,
    {new = 1, get = λi:Nat. i, inc = λi:Nat. succ(i),
     rcADT =
       {*Nat,
        {new = 1, get = λi:Nat. i, inc = λi:Nat. succ(i),
         reset = λi:Nat. 1}}
       as {∃ResetCounter<:Nat,
           {new: ResetCounter, get: ResetCounter→Nat,
            inc: ResetCounter→ResetCounter,
            reset: ResetCounter→ResetCounter}} }}
   as {∃Counter,
       {new: Counter, get: Counter→Nat, inc: Counter→Counter,
        rcADT:
          {∃ResetCounter<:Counter,
           {new: ResetCounter, get: ResetCounter→Nat,
            inc: ResetCounter→ResetCounter,
            reset: ResetCounter→ResetCounter}}}};
▸ counterADT : {∃Counter,
                 {new:Counter,get:Counter→Nat,inc:Counter→Counter,
                  rcADT:{∃ResetCounter<:Counter,
                          {new:ResetCounter,get:ResetCounter→Nat,
                           inc:ResetCounter→ResetCounter,
                           reset:ResetCounter→ResetCounter}}}}
```

これらのパッケージが開かれると、結果として、残りのプログラムが検査される文脈は `Counter<:Top`、`counter:{...}`、`ResetCounter<:Counter`、`resetCounter:{...}`、という形の型変数束縛（と項変数束縛）を持つ。

```
let {Counter,counter} = counterADT in
let {ResetCounter,resetCounter} = counter.rcADT in
counter.get
  (counter.inc
     (resetCounter.reset (resetCounter.inc resetCounter.new)));
▸ 2 : Nat
```

解答 26.5.4. 24.3 節の表現に上界を追加するだけでよい。追加すべき場所は明らかで、型のレベルでは、

$$\{\exists X<:S,T\} \stackrel{\text{def}}{=} \forall Y.\ (\forall X<:S.\ T\to Y) \to Y$$

とする。項のレベルにもこれに合わせた変更を行う。

解答 27.1. 一つの方法を挙げる。

```
setCounterClass =
  λM<:SetCounter. λR<:CounterRep.
  λself: Ref(R→M).
    λr: R.
       {get = λ_:Unit. !(r.x),
        set = λi:Nat.  r.x:=i,
        inc = λ_:Unit. (!self r).set (succ((!self r).get unit))};
▸ setCounterClass : ∀M<:SetCounter.
                       ∀R<:CounterRep.
                         (Ref (R→M)) → R → SetCounter

instrCounterClass =
  λM<:InstrCounter.
  λR<:InstrCounterRep.
  λself: Ref(R→M).
    λr: R.
      let super = setCounterClass [M] [R] self in
         {get = (super r).get,
          set = λi:Nat. (r.a:=succ(!(r.a)); (super r).set i),
          inc = (super r).inc,
          accesses = λ_:Unit. !(r.a)};
▸ instrCounterClass : ∀M<:InstrCounter.
                         ∀R<:InstrCounterRep.
                           (Ref (R→M)) → R → InstrCounter

newInstrCounter =
  let m = ref (λr:InstrCounterRep. error as InstrCounter) in
  let m' =
    instrCounterClass [InstrCounter] [InstrCounterRep] m in
  (m := m';
   λ_:Unit. let r = {x=ref 1, a=ref 0} in m' r);
▸ newInstrCounter : Unit → InstrCounter
```

解答 28.2.3. T-TAbs の場合、S-All の追加の前提を与えるため、単純に S-Refl を用いる。T-TApp の場合、部分型の逆転補題（フル $F_{<:}$ 版）から $N_1 = \forall X<:N_{11}.N_{12}$、ただし $\Gamma \vdash T_{11} <: N_{11}$ かつ $\Gamma, X<:T_{11} \vdash N_{12} <: T_{12}$ とわかる。推移律を用いて、$\Gamma \vdash T_2 <: N_{11}$ といえる。よって、TA-TApp を用いて $\Gamma \mapsto t_1\ [T_2] : [X \mapsto T_2]N_{12}$ が得られる。あとは以前と同じように、代入における部分型付けの保存（補題 26.4.8）より、$\Gamma \vdash [X \mapsto T_2]N_{12} <: [X \mapsto T_2]T_{12} = T$ が得られ、証明は完了する。

解答 28.5.1. 定理 28.3.5（特に S-All の場合）がフル $F_{<:}$ では失敗する。

解答 28.5.6. まず、有界量化子と非有界量化子を混ぜることは許されないことに注意されたい。二つの有界量化子を比較する部分型付け規則や、二つの非有界量化子を比較する規則はあるが、有界量化子と非有界量化子を比較する規則は存在してはならない。さもなくば振り出しに戻ってしまう。

$$R(\forall Y{<:}S.T) = \begin{cases} \forall Y{<:}S.\ R(T) & X \notin FV(S)\ \text{の場合} \\ \mathsf{Top} & X \in FV(S)\ \text{の場合} \end{cases}$$

$$L(\forall Y{<:}S.T) = \begin{cases} \forall Y{<:}S.\ L(T) & L(T) \neq \mathit{fail}\ \text{かつ}\ X \notin FV(S)\ \text{の場合} \\ \mathit{fail} & \text{それ以外の場合} \end{cases}$$

$$R(\{\exists Y{<:}S.T\}) = \begin{cases} \{\exists Y{<:}S, R(T)\} & X \notin FV(S)\ \text{の場合} \\ \mathsf{Top} & X \in FV(S)\ \text{の場合} \end{cases}$$

$$L(\{\exists Y{<:}S.T\}) = \begin{cases} \{\exists Y{<:}S.\ L(T)\} & L(T) \neq \mathit{fail}\ \text{かつ}\ X \notin FV(S)\ \text{の場合} \\ \mathit{fail} & \text{それ以外の場合} \end{cases}$$

$$R(S \to T) = \begin{cases} L(S) \to R(T) & L(S) \neq \mathit{fail}\ \text{の場合} \\ \mathsf{Top} & L(S) = \mathit{fail}\ \text{の場合} \end{cases}$$

$$L(S \to T) = \begin{cases} R(S) \to L(T) & L(T) \neq \mathit{fail}\ \text{の場合} \\ \mathit{fail} & L(T) = \mathit{fail}\ \text{の場合} \end{cases}$$

$$R(X) = T \quad \text{ただし}\ X{<:}T \in \Gamma \qquad\qquad L(X) = \mathit{fail}$$

$$R(Y) = Y \quad Y \neq X\ \text{の場合} \qquad\qquad L(Y) = Y \quad Y \neq X\ \text{の場合}$$

$$R(\mathsf{Top}) = \mathsf{Top} \qquad\qquad L(\mathsf{Top}) = \mathsf{Top}$$

▲ 図 A-2. 与えられた型の、X を含まない最小の上位型と最大の上位型

(1) と (2) については、詳しくは Katiyar and Sankar [251] を参照せよ。(3) については、「満足いくものではない」が答えである。幅部分型付けのあるレコード型を制限された型システムに追加すると、決定不能に戻ってしまう。問題は、空のレコード型が（レコード型の中での）最大の型のようなものであることである。これを用いると、Ghelli の例を修正したものによって、部分型検査器に発散を生じさせることができる。$T = \forall X{<:}\{\}.\ \neg\{a{:}\ \forall Y{<:}X.\ \neg Y\}$ としたとき、$X_0{<:}\{a{:}T\} \vdash X_0 <: \{a{:}\ \forall X_1{<:}X_0.\ \neg X_1\}$ という入力で部分型検査器を発散させられる。

この例を作るにあたっては Martin Hofmann が手助けしてくれた。Katiyar and Sankar [251] が同じ観察をしている。

解答 28.6.3.

(1) 九つの共通の部分型を挙げる。

$\forall X{<:}Y'\to Z.\ Y\to Z'$	$\forall X{<:}Y'\to Z.\ \mathsf{Top}\to Z'$	$\forall X{<:}Y'\to Z.\ X$
$\forall X{<:}Y'\to \mathsf{Top}.\ Y\to Z'$	$\forall X{<:}Y'\to \mathsf{Top}.\ \mathsf{Top}\to Z'$	$\forall X{<:}Y'\to \mathsf{Top}.\ X$
$\forall X{<:}\mathsf{Top}.\ Y\to Z'$	$\forall X{<:}\mathsf{Top}.\ \mathsf{Top}\to Z'$	$\forall X{<:}\mathsf{Top}.\ X$

(2) $\forall X{<:}Y'\to Z.\ Y\to Z'$ と $\forall X{<:}Y'\to Z.\ X$ はどちらも S と T の下界であるが、これら二つの型は、S と T の部分型であるような共通の上位型を持たない。

(3) $S\to\mathsf{Top}$ と $T\to\mathsf{Top}$ を考えよ。（または $\forall X{<:}Y'\to Z.\ Y\to Z'$ と $\forall X{<:}Y'\to Z.\ X$。）

解答 28.7.1. 関数 $R_{X,\Gamma}$ と $L_{X,\Gamma}$ を図 A-2 に定義する。どちらも型から、その型の X を含まない最小の上位型、最大の部分型にそれぞれ写す。（雑然とならないよう、下付き文字の X と Γ は省略している。）二つの定義の付帯条件は異なる。なぜなら、L が現れるときは、定義されているかどうか（$L(T) \neq \mathit{fail}$ と書く）を必ず検査しなければならないが、一方 R は Top 型があるおかげで常に定義されるためである。これらの定義の正しさは Ghelli and Pierce [175] で証明されている。

解答 28.7.2. フルの有界存在型の決定不能性を証明する簡単な方法（Ghelli and Pierce [175] による）は、フル $F_{<:}$ の部分型付け問題から存在型だけを持つ型システムにおける部分型付け問題への翻訳 $[\![-]\!]$ を与えることである。$\Gamma \vdash S <: T$ が $F_{<:}$ において証明可能なとき、かつそのときに限り、$[\![\Gamma \vdash S <: T]\!]$ が存在型を持つ型システムで証明可能になるように翻訳を定義する。この翻訳は型においては以下のように定義できる。

$$
\begin{array}{lcllcl}
[\![X]\!] & = & X & [\![\mathrm{Top}]\!] & = & \mathrm{Top} \\
[\![\forall X{<:}T_1.T_2]\!] & = & \neg\{\exists X{<:}T_1,\neg[\![T_2]\!]\} & [\![T_1 \to T_2]\!] & = & [\![T_1]\!] \to [\![T_2]\!]
\end{array}
$$

ただし $\neg S = S \to \mathrm{Top}$ とする[†11]。これを、$[\![X_1{<:}T_1, \ldots, X_n{<:}T_n]\!] = X_1{<:}[\![T_1]\!], \ldots, X_n{<:}[\![T_n]\!]$ というように文脈に対して拡張する。さらに、$[\![\Gamma \vdash S <: T]\!] = [\![\Gamma]\!] \vdash [\![S]\!] <: [\![T]\!]$ というように部分型付け関係に対して拡張する。

解答 29.1.1. $\forall X.X\to X$ は真の型であり、その要素は $\lambda X.\lambda x{:}X.x$ などとなる。これらの項は多相関数であり、型 T で具体化されると T から T への関数を返す。一方、$\lambda X.X\to X$ は型演算子である。すなわち、型 T に適用されると、T から T への関数を要素として持つ真の型 $T\to T$ を返す。

言い換えると、$\forall X.X\to X$ は、その要素が型から項への項レベル関数であるような型である。その要素を（t [T] と書いて型に適用することで）具体化すると、矢印型 $T\to T$ の**要素**を返す。一方、$\lambda X.X\to X$ は**それ自身が**（型から型への）関数である。この型自体を（$(\lambda X.X\to X)\ T$ と書いて）型 T で具体化すると、型 $T\to T$ **そのもの**を返すのであって、その要素を返すわけではない。

例えば、fn が型 $\forall X.X\to X$ を持ち、$\mathrm{Op} = \lambda X.X\to X$ とすると、$\mathrm{fn}\ [T] : T\to T = \mathrm{Op}\ T$ といえる。

解答 29.1.2. $\mathrm{Nat}\to\mathrm{Nat}$ は関数の（真の）型であり、型レベルの関数ではない。

解答 30.3.3. 補題 30.3.1 を用いるのは T-Abs、T-TApp、T-Eq の場合である。補題 30.3.2 を用いるのは T-Var の場合である。

解答 30.3.8. 与えられた二つの導出のサイズの合計に関する帰納法による。両方の導出の最後に用いられる規則に関する場合分けを行う。どちらかの導出が QR-Refl で終わる場合、もう一つの導出が所望の結論である。どちらかの導出が QR-Abs、QR-Arrow、QR-All で終わる場合、規則の形から、もう片方の導出も同じ規則で終わらなければならない。帰納法の仮定を単純に適用することで結論が得られる。両方の導出が QR-App で終わる場合、やはり帰納法の仮定を単純に適用して結論が得られる。残る場合はもっと興味深い。

両方の導出が QR-AppAbs で終わる場合、以下を得る。

$$S = (\lambda X{::}K_{11}.S_{12})\ S_2 \qquad T = [X \mapsto T_2]T_{12} \qquad U = [X \mapsto U_2]U_{12}$$

ただし

$$S_{12} \Rightarrow T_{12} \qquad S_2 \Rightarrow T_2 \qquad S_{12} \Rightarrow U_{12} \qquad S_2 \Rightarrow U_2$$

である。帰納法の仮定により、V_{12} と V_2 が存在して、

$$T_{12} \Rightarrow V_{12} \qquad T_2 \Rightarrow V_2 \qquad U_{12} \Rightarrow V_{12} \qquad U_2 \Rightarrow V_2$$

である。補題 30.3.7 を二回適用して、$[X \mapsto T_2]T_{12} \Rightarrow [X \mapsto V_2]V_{12}$ と $[X \mapsto U_2]U_{12} \Rightarrow [X \mapsto V_2]V_{12}$ を得られる。つまり、$T \Rightarrow V$ かつ $U \Rightarrow V$。

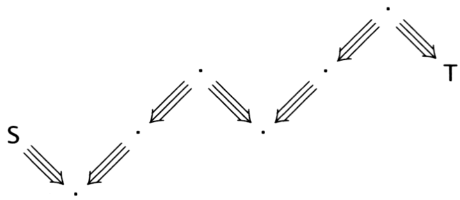
[†11]（訳注）この ¬ の定義は一般には正しくないが、ここでは反変であればよいので十分である。

最後に、片方の導出（一つ目のほうとしよう）がQR-AppPで終わり、もう片方の導出がQR-AppAbsで終わるとする。この場合は、以下を得る。

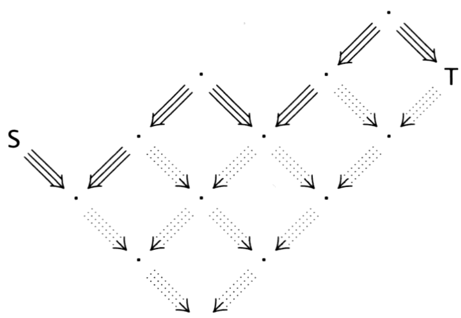
$$S = (\lambda X{::}K_{11}.S_{12})\ S_2 \qquad T = (\lambda X{::}K_{11}.T_{12})\ T_2 \qquad U = [X \mapsto U_2]U_{12}$$

ただし今回も $S_{12} \Rightarrow T_{12}$、$S_2 \Rightarrow T_2$、$S_{12} \Rightarrow U_{12}$、$S_2 \Rightarrow U_2$ である。やはり帰納法の仮定により、V_{12} と V_2 が存在して $T_{12} \Rightarrow V_{12}$、$T_2 \Rightarrow V_2$、$U_{12} \Rightarrow V_{12}$、$U_2 \Rightarrow V_2$ である。この一つ目と二つ目に規則QR-AppAbsを適用し、三つ目と四つ目に補題30.3.7を適用すると、$T \Rightarrow V$ かつ $U \Rightarrow V$ とわかる。

解答 30.3.10. 最初の観察は、$S \Leftrightarrow^* T$ の任意の導出が、対称律の規則のインスタンスの部分導出において対称律も推移律も使われないように再構成できるということである。つまり、単一ステップの簡約の後にゼロ個または一個の対称律のインスタンスが続くものをQR-Transで張り合わせた列により、SからTに辿り着くことができる。この列は以下のように可視化できる。



（右から左へ向いた矢印は、対称律で終わる簡約であり、左から右への矢印は対称律が用いられていない簡約である。）ここで補題30.3.8を繰り返し用いて、この図の一番下がSとTの共通の簡約結果に到達するまで小さなダイヤモンドを追加する。



同様の議論を、図を使わず標準的な帰納法の形で示すこともできるが、おそらく理解しにくくなるだけで説得力が増すことはないだろう。

解答 30.3.17. 一つ目の妙な規則を追加すると、進行の性質が壊れるが、保存は成り立つ。二つ目の規則を追加すると、進行も保存も壊れる。

解答 30.3.20. あなたの解答と fomega 検査器のソースコードを比べてみよ。

解答 30.5.1. 型の族 FloatList n の代わりに、今回は**多相**型の族 List T n を用いて、以下のような演算を得る。

```
nil   :  ∀X. List X 0
cons  :  ∀X. Πn:Nat. X → List X (succ n)
hd    :  ∀X. Πn:Nat. List X (succ n) → X
tl    :  ∀X. Πn:Nat. List X (succ n) → List X n
```

解答 31.2.1.

Γ	⊢	A	<:	Id B	**導出可能**
Γ	⊢	Id A	<:	B	**導出可能**
Γ	⊢	λX.X	<:	λX.Top	**導出可能**
Γ	⊢	λX. ∀Y<:X. Y	<:	λX. ∀Y<:Top. Y	**導出不可能**
Γ	⊢	λX. ∀Y<:X. Y	<:	λX. ∀Y<:X. X	**導出可能**
Γ	⊢	F B	<:	B	**導出可能**
Γ	⊢	B	<:	F B	**導出不可能**
Γ	⊢	F A	<:	F B	**導出不可能**
Γ	⊢	∀G<:(λY.Top→Y). G A	<:	∀G<:(λY.Top→Y). Top→B	**導出可能**
Γ	⊢	∀G<:(λY.Top→Y). G A	<:	∀G<:(λY.Top→Y). G B	**導出不可能**
Γ	⊢	Top[*⇒*]	<:	Top[*⇒*⇒*]	**導出不可能**

解答 32.5.1. 重要な観察は、Object M が存在型だということである。Object は以下の演算子の省略記法である。

```
λM::*⇒*. {∃X, {state:X, methods:M X}}
```

これを M に適用すると簡約基となり、簡約すると以下の存在型になる。

```
{∃X, {state:X, methods:M X}}
```

この変形に包摂は現れない（よって情報損失も起きない）ことに注意されたい。

解答 32.5.2.

```
sendget =
  λM<:CounterM. λo:Object M.
    let {X, b} = o in b.methods.get(b.state);

sendreset =
  λM<:ResetCounterM. λo:Object M.
    let {X, b} = o in
      {*X,
       {state = b.methods.reset(b.state),
        methods = b.methods}}  as Object M;
```

解答 32.7.2. このように定義した計算体系では、最小型付けの性質は成り立たない。以下の項を考えよ。

```
{#x={a=5,b=7}}
```

この項には {#x:{a:Nat}} と {#x:{a:Nat,b:Nat}} のどちらの型も与えられるが、これらの型同士は比較不能である。これを合理的に修正する一つの方法は、レコードの項における不変のフィールドすべてに対して、意図する型を明示的に注釈することである。これは実質的に、上記二つの型のどちらかを選ぶ責任をプログラマに押し付けている。

解答 32.9.1.

```
MyCounterM =
  λR. {get: R→Nat, set:R→Nat→R, inc:R→R, accesses:R→Nat,
             backup:R→R, reset:R→R};

MyCounterR = {#x:Nat,#count:Nat,#old:Nat};

myCounterClass =
   λR<:MyCounterR.
   λself: Unit→MyCounterM R.
   λ_:Unit.
      let super = instrCounterClass [R] self unit in
      {get = super.get,
       set = super.set,
       inc = super.inc,
       accesses = super.accesses,
       reset = λs:R. s←x=s.old,
       backup = λs:R. s←old=s.x}
   as MyCounterM R;

mc = {*MyCounterR,
       {state = {#x=0,#count=0,#old=0},
        methods = fix (myCounterClass [MyCounterR]) unit}}
      as Object MyCounterM;

sendget [MyCounterM]
  (sendreset [MyCounterM] (sendinc [MyCounterM] mc));
```

「ワトソン君、少しは自分で分析してみたまえ」ホームズは苛立たしく言った。「僕のやり方は知ってるはずだ。その通りにやりたまえ。そうして互いの結果を比べれば教育的というもんだ」

—*A. Conan Doyle,* 四つの署名*, 1890*

これを完璧に説明する方法については、読者への演習としておく。これは、数学者が説明の改訂をせずに済ませるための常套手段である。

—*W. v. O. Quine, 1987* [393]

記法

B.1 メタ変数名

本文	ML コード	用途
p, q, r, s, t, u	s, t	項
x, y, z	x, y	項変数
v, w	v, w	値
nv	nv	数値
l, j, k	l	レコードとバリアントのフィールドのラベル
μ	store	ストア
M, N, P, Q, S, T, U, V	tyS, tyT	型
A, B, C	tyA, tyB	基本型
Σ		ストア型付け
X, Y, Z	tyX, tyY	型変数
K, L	kK, kL	カインド
σ		代入
Γ, Δ	ctx	文脈
$\mathcal{J}$		任意の言明
$\mathcal{D}$		型付け導出
$\mathcal{C}$		部分型付け導出
	fi	ファイル中の位置情報
i, j, k, l		数値添字

B.2 規則名の規約

接頭辞	用途
B-	大ステップ評価
CT-	制約型付け
E-	評価
K-	カインド付け
M-	マッチング
P-	パターン型付け
Q-	型等価性

接頭辞	用途
QR-	型の並行簡約
S-	部分型付け
SA-	アルゴリズム的部分型付け
T-	型付け
TA-	アルゴリズム的型付け
XA-	型露出

B.3 名付けと添字の規約

本書におけるメタ変数名、数値添字、プライム記号の選択は、以下の原則に従っている。

(1) 構文定義では、裸のメタ変数 t をすべての項に用い、同様に T を型に、v を値に用いるなどとする。
(2) 型付け規則では、主となる項（その型を計算する項）を常に t と呼び、その部分項を t_1、t_2、…と名付ける。（ときどき（例えば、簡約規則で）、部分項の部分項が必要になることがある。その際には t_{11}、t_{12}、…などを用いる。）
(3) 評価規則では、簡約対象全体の項を t と呼び、簡約結果の項を t′ と呼ぶ。
(4) 項 t の型は T と呼ぶ。（同様に、部分項 t_1 の型は T_1 などとする。）
(5) 定理を述べる際や証明する際にも同じ規約を用いる。ただし、定義と定理の間での名前の衝突を避けるために、t を s に（同様に T を S や R に）置き換えることがある。

少数ながら、これらの規則を同時に満足できない場合もある。その場合には、先に現れる規則を優先する。（例えば、図 11-5 の T-PROJ1 規則では、規則 (4) を緩和しており、部分項 t_1 の型が $T_{11} \times T_{12}$ となっている。）また、ごく少数の箇所（例えば、図 11-7 のレコード射影規則 T-PROJ）ではこれら規則を完全に無視している。これら規則に従うと、受け入れがたいほど醜くなるか読みにくくなるためである。

参考文献

[1] Abadi, Martín. Secrecy by typing in security protocols. *Journal of the ACM*, 46(5):749–786, September 1999. Summary in *Theoretical Aspects of Computer Software (TACS), Sendai, Japan*, 1997; volume 1281 of Springer LNCS.

[2] Abadi, Martín, Anindya Banerjee, Nevin Heintze, and Jon G. Riecke. A core calculus of dependency. In *ACM Symposium on Principles of Programming Languages (POPL), San Antonio, Texas*, pages 147–160, 1999.

[3] Abadi, Martín and Luca Cardelli. On subtyping and matching. In *European Conference on Object-Oriented Programming (ECOOP)*, pages 145–167, 1995.

[4] Abadi, Martín and Luca Cardelli. *A Theory of Objects*. Springer-Verlag, 1996.

[5] Abadi, Martín, Luca Cardelli, and Pierre-Louis Curien. Formal parametric polymorphism. *Theoretical Computer Science*, 121(1–2):9–58, 6 December 1993. Summary in *ACM Symposium on Principles of Programming Languages (POPL), Charleston, South Carolina*, 1993.

[6] Abadi, Martín, Luca Cardelli, Pierre-Louis Curien, and Jean-Jacques Lévy. Explicit substitutions. *Journal of Functional Programming*, 1(4):375–416, 1991. Summary in *ACM Symposium on Principles of Programming Languages (POPL), San Francisco, California*, 1990.

[7] Abadi, Martín, Luca Cardelli, Benjamin Pierce, and Gordon Plotkin. Dynamic typing in a statically typed language. *ACM Transactions on Programming Languages and Systems*, 13(2):237–268, April 1991. Summary in *ACM Symposium on Principles of Programming Languages (POPL), Austin, Texas*, 1989.

[8] Abadi, Martín, Luca Cardelli, Benjamin Pierce, and Didier Rémy. Dynamic typing in polymorphic languages. *Journal of Functional Programming*, 5(1):111–130, January 1995. Summary in *ACM SIGPLAN Workshop on ML and its Applications*, June 1992.

[9] Abadi, Martín, Luca Cardelli, and Ramesh Viswanathan. An interpretation of objects and object types. In *ACM Symposium on Principles of Programming Languages (POPL), St. Petersburg Beach, Florida*, pages 396–409, 1996.

[10] Abadi, Martín and Marcelo P. Fiore. Syntactic considerations on recursive types. In *Proceedings of the 11th Annual IEEE Symposium on Logic in Computer Science, LICS '96*, pages 242–252. IEEE Computer Society Press, Los Alamitos, CA, July 1996.

[11] Abelson, Harold and Gerald Sussman. *Structure and Interpretation of Computer Programs*. MIT Press, New York, 1985. Second edition, 1996.

[12] Abramsky, Samson, Radha Jagadeesan, and Pasquale Malacaria. Full abstraction for PCF. *Information and Computation*, 163(2):409–470, December 2000.

[13] Aczel, Peter. An introduction to inductive definitions. In Jon Barwise, editor, *Handbook of Mathematical Logic*, number 90 in Studies in Logic and the Foundations of Mathematics, pages 739–782. North Holland, 1977.

[14] Aczel, Peter. *Non-Well-Founded Sets*. Stanford Center for the Study of Language and Information, 1988. CSLI Lecture Notes number 14.

[15] Agesen, Ole, Stephen N. Freund, and John C. Mitchell. Adding type parameterization to the Java language. In *ACM Symposium on Object Oriented Programming: Systems, Languages, and Applications (OOPSLA)*, pages 49–65, Atlanta, GA, October 1997.

[16] Aho, Alfred V., Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools.* Addison-Wesley, Reading, MA, USA, 1986.

[17] Aiken, Alexander and Edward L. Wimmers. Type inclusion constraints and type inference. In *ACM Symposium on Functional Programming Languages and Computer Architecture (FPCA)*, pages 31–41, 1993.

[18] Amadio, Roberto M. and Luca Cardelli. Subtyping recursive types. *ACM Transactions on Programming Languages and Systems*, 15(4):575–631, 1993. Summary in *ACM Symposium on Principles of Programming Languages (POPL), Orlando, Florida*, pp. 104–118; also DEC/Compaq Systems Research Center Research Report number 62, August 1990.

[19] Appel, Andrew W. *Modern Compiler Implementation in ML.* Cambridge University Press, 1998.

[20] Appel, Andrew W. and Marcelo J. R. Gonçalves. Hash-consing garbage collection. Technical Report CS-TR-412-93, Princeton University, Computer Science Department, 1993.

[21] Arbib, Michael and Ernest Manes. *Arrows, Structures, and Functors: The Categorical Imperative.* Academic Press, 1975.

[22] Ariola, Zena M., Matthias Felleisen, John Maraist, Martin Odersky, and Philip Wadler. A call-by-need lambda calculus. In *ACM Symposium on Principles of Programming Languages (POPL), San Francisco, California*, pages 233–246, January 1995.

[23] Arnold, Ken and James Gosling. *The Java Programming Language.* Addison Wesley, 1996.

[24] Arnold, Ken, Ann Wollrath, Bryan O'Sullivan, Robert Scheifler, and Jim Waldo. *The Jini specification.* Addison-Wesley, Reading, MA, USA, 1999.

[25] Asperti, Andrea and Giuseppe Longo. *Categories, Types, and Structures: An Introduction to Category Theory for the Working Computer Scientist.* MIT Press, 1991.

[26] Aspinall, David. Subtyping with singleton types. In *Computer Science Logic (CSL), Kazimierz, Poland*, pages 1–15. Springer-Verlag, 1994.

[27] Aspinall, David and Adriana Compagnoni. Subtyping dependent types. *Information and Computation*, 266(1–2):273–309, September 2001. Preliminary version in *IEEE Symposium on Logic in Computer Science (LICS)*, 1996.

[28] Astesiano, Egidio. Inductive and operational semantics. In E. J. Neuhold and M. Paul, editors, *Formal Description of Programming Concepts*, IFIP State-of-the-Art Reports, pages 51–136. Springer-Verlag, 1991.

[29] Augustsson, Lennart. A compiler for Lazy ML. In *ACM Symposium on Lisp and Functional Programming (LFP), Austin, Texas*, pages 218–227, August 1984.

[30] Augustsson, Lennart. Cayenne — a language with dependent types. In *International Conference on Functional Programming (ICFP), Baltimore, Maryland, USA*, pages 239–250, 1998.

[31] Baader, Franz and Tobias Nipkow. *Term Rewriting and All That.* Cambridge University

Press, 1998.

[32] Baader, Franz and Jörg Siekmann. Unification theory. In D. M. Gabbay, C. J. Hogger, and J. A. Robinson, editors, *Handbook of Logic in Artificial Intelligence and Logic Programming*, volume 2, Deduction Methodologies, pages 41–125. Oxford University Press, Oxford, UK, 1994.

[33] Backus, John. Can programming be liberated from the von Neumann style? A functional style and its algebra of programs. *Communications of the ACM*, 21(8): 613–641, August 1978. Reproduced in S. S. Thakkar, editor, *Selected Reprints on Dataflow and Reduction Architectures*, IEEE, 1987, pp. 215–243, and in *ACM Turing Award Lectures: The First Twenty Years*, ACM Press, 1987, pp. 63–130.

[34] Backus, John. The history of Fortran I, II, and III. In Wexelblat, editor, *History of Programming Languages*, pages 25–45. Academic Press, 1981.

[35] Bainbridge, E. Stewart, Peter J. Freyd, Andre Scedrov, and Philip J. Scott. Functorial polymorphism. *Theoretical Computer Science*, 70(1):35–64, 1990. Corrigendum in *TCS* 71(3), 431.

[36] Baldan, Paolo, Giorgio Ghelli, and Alessandra Raffaetà. Basic theory of F-bounded quantification. *Information and Computation*, 153(1):173–237, 1999.

[37] Barendregt, Henk P. *The Lambda Calculus*. North Holland, revised edition, 1984.

[38] Barendregt, Henk P. Functional programming and lambda calculus. In Jan van Leeuwen, editor, *Handbook of Theoretical Computer Science, Volume B*, chapter 7, pages 321–364. Elsevier / MIT Press, 1990.

[39] Barendregt, Henk P. Introduction to generalized type systems. *Journal of Functional Programming*, 1(2):125–154, 1991.

[40] Barendregt, Henk P. Lambda calculi with types. In Abramsky, Gabbay, and Maibaum, editors, *Handbook of Logic in Computer Science*, volume II. Oxford University Press, 1992.

[41] Barras, Bruno, Samuel Boutin, Cristina Cornes, Judicael Courant, Jean-Christophe Filliatre, Eduardo Gimenez, Hugo Herbelin, Gerard Huet, Cesar Munoz, Chetan Murthy, Catherine Parent, Christine Paulin-Mohring, Amokrane Saibi, and Benjamin Werner. The Coq proof assistant reference manual : Version 6.1. Technical Report RT-0203, Inria (Institut National de Recherche en Informatique et en Automatique), France, 1997.

[42] Barwise, Jon and Lawrence Moss. *Vicious Circles: On the Mathematics of Non-wellfounded Phenomena*. Cambridge University Press, 1996.

[43] Berardi, Stefano. Towards a mathematical analysis of the Coquand-Huet calculus of constructions and the other systems in Barendregt's cube. Technical report, Department of Computer Science, CMU, and Dipartimento Matematica, Universita di Torino, 1988.

[44] Berger, Ulrich. Program extraction from normalization proofs. In Marc Bezem and Jan Friso Groote, editors, *Typed Lambda Calculi and Applications*, number 664 in Lecture Notes in Computer Science, pages 91–106, Utrecht, The Netherlands, March 1993. Springer-Verlag.

[45] Berger, Ulrich and Helmut Schwichtenberg. An inverse of the evaluation functional for typed λ-calculus. In Gilles Kahn, editor, *IEEE Symposium on Logic in Computer Science*

(LICS), pages 203–211, Amsterdam, The Netherlands, July 1991. IEEE Computer Society Press.

[46] Birtwistle, Graham M., Ole-Johan Dahl, Bjorn Myhrhaug, and Kristen Nygaard. *Simula Begin.* Studentlitteratur (Lund, Sweden), Bratt Institut fuer neues Lernen (Goch, FRG), Chartwell-Bratt Ltd (Kent, England), 1979.

[47] Bobrow, Daniel G., Linda G. DeMichiel, Richard P. Gabriel, Sonya E. Keene, Gregor Kiczales, and David A. Moon. Common Lisp Object System specification X3J13 document 88-002R. *SIGPLAN Notices*, 23, 1988.

[48] Boehm, Hans-J. Partial polymorphic type inference is undecidable. In *26th Annual Symposium on Foundations of Computer Science*, pages 339–345. IEEE, October 1985.

[49] Boehm, Hans-J. Type inference in the presence of type abstraction. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), Portland, Oregon*, pages 192–206, June 1989.

[50] Böhm, Corrado and Alessandro Berarducci. Automatic synthesis of typed Λ-programs on term algebras. *Theoretical Computer Science*, 39(2–3):135–154, August 1985.

[51] Bono, Viviana and Kathleen Fisher. An imperative first-order calculus with object extension. In *European Conference on Object-Oriented Programming (ECOOP)*, 1998.

[52] Bono, Viviana, Amit J. Patel, and Vitaly Shmatikov. A core calculus of classes and mixins. In *European Conference on Object-Oriented Programming (ECOOP)*, volume 1628 of *Lecture Notes in Computer Science*, pages 43–66. Springer-Verlag, June 1999.

[53] Bono, Viviana, Amit J. Patel, Vitaly Shmatikov, and John C. Mitchell. A core calculus of classes and objects. In *Fifteenth Conference on the Mathematical Foundations of Programming Semantics*, April 1999.

[54] Bracha, Gilad, Martin Odersky, David Stoutamire, and Philip Wadler. Making the future safe for the past: Adding genericity to the Java programming language. In Craig Chambers, editor, *ACM Symposium on Object Oriented Programming: Systems, Languages, and Applications (OOPSLA)*, ACM SIGPLAN Notices volume 33 number 10, pages 183–200, Vancouver, BC, October 1998.

[55] Braithwaite, Richard B. *The Foundations of Mathematics: Collected Papers of Frank P. Ramsey.* Routledge and Kegan Paul, London, 1931.

[56] Brandt, Michael and Fritz Henglein. Coinductive axiomatization of recursive type equality and subtyping. In Roger Hindley, editor, *Proc. 3rd Int'l Conf. on Typed Lambda Calculi and Applications (TLCA), Nancy, France, April 2–4, 1997*, volume 1210 of *Lecture Notes in Computer Science (LNCS)*, pages 63–81. Springer-Verlag, April 1997. Full version in Fundamenta Informaticae, Vol. 33, pp. 309–338, 1998.

[57] Breazu-Tannen, Val, Thierry Coquand, Carl Gunter, and Andre Scedrov. Inheritance as implicit coercion. *Information and Computation*, 93:172–221, 1991. Also in Carl A. Gunter and John C. Mitchell, editors, *Theoretical Aspects of Object-Oriented Programming: Types, Semantics, and Language Design* (MIT Press, 1994).

[58] Bruce, Kim B. The equivalence of two semantic definitions for inheritance in object-oriented languages. In *Proceedings of Mathematical Foundations of Programming Semantics*, Pittsburgh, PA, March 1991.

[59] Bruce, Kim B. A paradigmatic object-oriented programming language: Design, static

typing and semantics. *Journal of Functional Programming*, 4(2), April 1994. Summary in *ACM Symposium on Principles of Programming Languages (POPL), Charleston, South Carolina*, under the title "Safe type checking in a statically typed object-oriented programming language".

[60] Bruce, Kim B. *Foundations of Object-Oriented Languages: Types and Semantics.* MIT Press, 2002.

[61] Bruce, Kim B., Luca Cardelli, Giuseppe Castagna, the Hopkins Objects Group (Jonathan Eifrig, Scott Smith, Valery Trifonov), Gary T. Leavens, and Benjamin Pierce. On binary methods. *Theory and Practice of Object Systems*, 1(3):221–242, 1996.

[62] Bruce, Kim B., Luca Cardelli, and Benjamin C. Pierce. Comparing object encodings. In *Theoretical Aspects of Computer Software (TACS), Sendai, Japan*, September 1997. An earlier version was presented as an invited lecture at the Third International Workshop on Foundations of Object Oriented Languages (FOOL 3), July 1996.

[63] Bruce, Kim B., Luca Cardelli, and Benjamin C. Pierce. Comparing object encodings. *Information and Computation*, 155(1/2):108–133, November 1999. Special issue of papers from *Theoretical Aspects of Computer Software (TACS 1997)*. An earlier version appeared as an invited lecture in the Third International Workshop on Foundations of Object Oriented Languages (FOOL 3), July 1996.

[64] Bruce, Kim B. and Giuseppe Longo. A modest model of records, inheritance, and bounded quantification. *Information and Computation*, 87:196–240, 1990. Also in Carl A. Gunter and John C. Mitchell, editors, *Theoretical Aspects of Object-Oriented Programming: Types, Semantics, and Language Design* (MIT Press, 1994). An earlier version appeared in the proceedings of the IEEE Symposium on Logic in Computer Science, 1988.

[65] Bruce, Kim B. and John Mitchell. PER models of subtyping, recursive types and higher-order polymorphism. In *ACM Symposium on Principles of Programming Languages (POPL), Albuquerque, New Mexico*, January 1992.

[66] Bruce, Kim B., Leaf Petersen, and Adrian Fiech. Subtyping is not a good "match" for object-oriented languages. In *European Conference on Object-Oriented Programming (ECOOP)*, volume 1241 of *Lecture Notes in Computer Science*, pages 104–127. Springer-Verlag, 1997.

[67] Buneman, Peter and Benjamin Pierce. Union types for semistructured data. In *Internet Programming Languages*. Springer-Verlag, September 1998. Proceedings of the International Database Programming Languages Workshop. LNCS 1686.

[68] Burstall, Rod and Butler Lampson. A kernel language for abstract data types and modules. In G. Kahn, D. MacQueen, and G. Plotkin, editors, *Semantics of Data Types*, volume 173 of *Lecture Notes in Computer Science*, pages 1–50. Springer-Verlag, 1984.

[69] Burstall, Rod M. Proving properties of programs by structural induction. *The Computer Journal*, 12(1):41–48, 1969.

[70] Canning, Peter, William Cook, Walter Hill, and Walter Olthoff. Interfaces for strongly-typed object-oriented programming. In *ACM Symposium on Object Oriented Programming: Systems, Languages, and Applications (OOPSLA)*, pages 457–467, 1989.

[71] Canning, Peter, William Cook, Walter Hill, Walter Olthoff, and John Mitchell. F-

bounded polymorphism for object-oriented programming. In *ACM Symposium on Functional Programming Languages and Computer Architecture (FPCA)*, pages 273–280, September 1989.

[72] Canning, Peter, Walt Hill, and Walter Olthoff. A kernel language for object-oriented programming. Technical Report STL-88-21, Hewlett-Packard Labs, 1988.

[73] Cardelli, Luca. A semantics of multiple inheritance. In G. Kahn, D. MacQueen, and G. Plotkin, editors, *Semantics of Data Types*, volume 173 of *Lecture Notes in Computer Science*, pages 51–67. Springer-Verlag, 1984. Full version in *Information and Computation*, 76(2/3):138–164, 1988.

[74] Cardelli, Luca. Amber. In Guy Cousineau, Pierre-Louis Curien, and Bernard Robinet, editors, *Combinators and Functional Programming Languages*, pages 21–47. Springer-Verlag, 1986. Lecture Notes in Computer Science No. 242.

[75] Cardelli, Luca. Basic polymorphic typechecking. *Science of Computer Programming*, 8 (2):147–172, April 1987. An earlier version appeared in the *Polymorphism Newsletter*, January, 1985.

[76] Cardelli, Luca. Structural subtyping and the notion of power type. In *ACM Symposium on Principles of Programming Languages (POPL), San Diego, California*, pages 70–79, January 1988.

[77] Cardelli, Luca. Typechecking dependent types and subtypes. In M. Boscarol, L. Carlucci Aiello, and G. Levi, editors, *Foundations of Logic and Functional Programming, Workshop Proceedings, Trento, Italy, (Dec. 1986)*, volume 306 of *Lecture Notes in Computer Science*, pages 45–57. Springer-Verlag, 1988.

[78] Cardelli, Luca. Notes about $F^{\omega}_{<:}$. Unpublished manuscript, October 1990.

[79] Cardelli, Luca. Typeful programming. In E. J. Neuhold and M. Paul, editors, *Formal Description of Programming Concepts*. Springer-Verlag, 1991. An earlier version appeared as DEC/Compaq Systems Research Center Research Report #45, February 1989.

[80] Cardelli, Luca. Extensible records in a pure calculus of subtyping. Research report 81, DEC/Compaq Systems Research Center, January 1992. Also in Carl A. Gunter and John C. Mitchell, editors, *Theoretical Aspects of Object-Oriented Programming: Types, Semantics, and Language Design* (MIT Press, 1994).

[81] Cardelli, Luca. An implementation of $F_{<:}$. Research report 97, DEC/Compaq Systems Research Center, February 1993.

[82] Cardelli, Luca. Type systems. In Allen B. Tucker, editor, *Handbook of Computer Science and Engineering*. CRC Press, 1996.

[83] Cardelli, Luca, James Donahue, Lucille Glassman, Mick Jordan, Bill Kalsow, and Greg Nelson. Modula-3 report (revised). Research report 52, DEC/Compaq Systems Research Center, November 1989.

[84] Cardelli, Luca and Xavier Leroy. Abstract types and the dot notation. In *Proceedings of the IFIP TC2 Working Conference on Programming Concepts and Methods*. North Holland, 1990. Also appeared as DEC/Compaq SRC technical report 56.

[85] Cardelli, Luca and Giuseppe Longo. A semantic basis for Quest. *Journal of Functional Programming*, 1(4):417–458, October 1991. Summary in ACM Conference on Lisp and

Functional Programming, June 1990. Also available as DEC/Compaq SRC Research Report 55, Feb. 1990.

[86] Cardelli, Luca, Simone Martini, John C. Mitchell, and Andre Scedrov. An extension of System F with subtyping. *Information and Computation*, 109(1-2):4-56, 1994. Summary in TACS '91 (Sendai, Japan, pp. 750-770).

[87] Cardelli, Luca and John Mitchell. Operations on records. *Mathematical Structures in Computer Science*, 1:3-48, 1991. Also in Carl A. Gunter and John C. Mitchell, editors, *Theoretical Aspects of Object-Oriented Programming: Types, Semantics, and Language Design* (MIT Press, 1994); available as DEC/Compaq Systems Research Center Research Report #48, August, 1989, and in the proceedings of MFPS '89, Springer LNCS volume 442.

[88] Cardelli, Luca and Peter Wegner. On understanding types, data abstraction, and polymorphism. *Computing Surveys*, 17(4):471-522, December 1985.

[89] Cardone, Felice. Relational semantics for recursive types and bounded quantification. In *Proceedings of the Sixteenth International Colloquium on Automata, Languages, and Programming*, volume 372 of *Lecture Notes in Computer Science*, pages 164-178, Stresa, Italy, July 1989. Springer-Verlag.

[90] Cardone, Felice and Mario Coppo. Type inference with recursive types: Syntax and semantics. *Information and Computation*, 92(1):48-80, 1991.

[91] Cartwright, Robert and Guy L. Steele, Jr. Compatible genericity with run-time types for the Java programming language. In Craig Chambers, editor, *ACM Symposium on Object Oriented Programming: Systems, Languages, and Applications (OOPSLA), Vancouver, British Columbia*, SIGPLAN Notices 33(10), pages 201-215. ACM, October 1998.

[92] Castagna, Giuseppe. *Object-Oriented Programming: A Unified Foundation*. Springer-Verlag, 1997.

[93] Castagna, Giuseppe, Giorgio Ghelli, and Giuseppe Longo. A calculus for overloaded functions with subtyping. *Information and Computation*, 117(1):115-135, 15 February 1995. preliminary version in LISP and Functional Programming, July 1992 (pp. 182-192), and as Rapport de Recherche LIENS-92-4, Ecole Normale Supérieure, Paris.

[94] Chambers, Craig. Object-oriented multi-methods in Cecil. In *European Conference on Object-Oriented Programming (ECOOP)*, pages 33-56, 1992.

[95] Chambers, Craig. The Cecil language: Specification and rationale. Technical report, University of Washington, March 1993.

[96] Chambers, Craig and Gary Leavens. Type-checking and modules for multi-methods. In *ACM Symposium on Object Oriented Programming: Systems, Languages, and Applications (OOPSLA)*, October 1994. SIGPLAN Notices 29(10).

[97] Chen, Gang and Giuseppe Longo. Subtyping parametric and dependent types. In Kamareddine et al., editors, *Type Theory and Term Rewriting*, September 1996. Invited lecture.

[98] Chirimar, Jawahar, Carl A. Gunter, and Jon G. Riecke. Reference counting as a computational interpretation of linear logic. *Journal of Functional Programming*, 6 (2):195-244, March 1996.

[99] Church, Alonzo. An unsolvable problem of elementary number theory. *American*

Journal of Mathematics, 58:354–363, 1936.

[100] Church, Alonzo. A formulation of the simple theory of types. *Journal of Symbolic Logic*, 5:56–68, 1940.

[101] Church, Alonzo. *The Calculi of Lambda Conversion.* Princeton University Press, 1941.

[102] Clement, Dominique, Joelle Despeyroux, Thierry Despeyroux, and Gilles Kahn. A simple applicative language: Mini-ML. In *ACM Conference on LISP and Functional Programming*, pages 13–27, 1986.

[103] Clinger, William, Daniel P. Friedman, and Mitchell Wand. A scheme for a higher-level semantic algebra. In John Reynolds and Maurice Nivat, editors, *Algebraic Methods in Semantics*, pages 237–250. Cambridge University Press, 1985.

[104] Colazzo, Dario and Giorgio Ghelli. Subtyping recursive types in Kernel Fun. In *14th Symposium on Logic in Computer Science (LICS'99)*, pages 137–146. IEEE, July 1999.

[105] Compagnoni, Adriana and Healfdene Goguen. Decidability of higher-order subtyping via logical relations, December 1997. Manuscript, available at `ftp://www.dcs.ed.ac.uk/pub/hhg/hosdec.ps.gz`.

[106] Compagnoni, Adriana and Healfdene Goguen. Typed operational semantics for higher order subtyping. Technical Report ECS-LFCS-97-361, University of Edinburgh, July 1997.

[107] Compagnoni, Adriana B. Decidability of higher-order subtyping with intersection types. In *Computer Science Logic*, September 1994. Kazimierz, Poland. Springer *Lecture Notes in Computer Science* 933, June 1995. Also available as University of Edinburgh, LFCS technical report ECS-LFCS-94-281, titled "Subtyping in $F^{\omega}_{\wedge}$ is decidable".

[108] Compagnoni, Adriana B. and Benjamin C. Pierce. Intersection types and multiple inheritance. *Mathematical Structures in Computer Science*, 6(5):469–501, October 1996. Preliminary version available as University of Edinburgh technical report ECS-LFCS-93-275 and Catholic University Nijmegen computer science technical report 93-18, Aug. 1993, under the title "Multiple Inheritance via Intersection Types".

[109] Constable, Robert L. Types in computer science, philosophy, and logic. In Samuel R. Buss, editor, *Handbook of Proof Theory*, volume 137 of *Studies in logic and the foundations of mathematics*, pages 683–786. Elsevier, 1998.

[110] Constable et al., Robert L. *Implementing Mathematics with the NuPRL Proof Development System.* Prentice–Hall, Englewood Cliffs, NJ, 1986.

[111] Cook, William. Object-oriented programming versus abstract data types. In J. W. de Bakker et al., editors, *Foundations of Object-Oriented Languages*, volume 489 of *Lecture Notes in Computer Science*, pages 151–178. Springer-Verlag, 1991.

[112] Cook, William and Jens Palsberg. A denotational semantics of inheritance and its correctness. In *ACM Symposium on Object Oriented Programming: Systems, Languages, and Applications (OOPSLA)*, pages 433–444, 1989.

[113] Cook, William R. *A Denotational Semantics of Inheritance.* PhD thesis, Brown University, 1989.

[114] Cook, William R., Walter L. Hill, and Peter S. Canning. Inheritance is not subtyping. In *ACM Symposium on Principles of Programming Languages (POPL), San Francisco, California*, pages 125–135, January 1990. Also in Carl A. Gunter and John C. Mitchell,

editors, *Theoretical Aspects of Object-Oriented Programming: Types, Semantics, and Language Design* (MIT Press, 1994).

[115] Coppo, Mario and Mariangiola Dezani-Ciancaglini. A new type-assignment for λ-terms. *Archiv Math. Logik*, 19:139–156, 1978.

[116] Coppo, Mario, Mariangiola Dezani-Ciancaglini, and Patrick Sallé. Functional characterization of some semantic equalities inside λ-calculus. In Hermann A. Maurer, editor, *Proceedings of the 6th Colloquium on Automata, Languages and Programming*, volume 71 of *LNCS*, pages 133–146, Graz, Austria, July 1979. Springer.

[117] Coquand, Thierry. *Une Théorie des Constructions.* PhD thesis, University Paris VII, January 1985.

[118] Coquand, Thierry and Gérard Huet. The Calculus of Constructions. *Information and Computation*, 76(2/3):95–120, February/March 1988.

[119] Courcelle, Bruno. Fundamental properties of infinite trees. *Theoretical Computer Science*, 25:95–169, 1983.

[120] Cousineau, Guy and Michel Mauny. *The Functional Approach to Programming.* Cambridge University Press, 1998.

[121] Crary, Karl. Sound and complete elimination of singleton kinds. Technical Report CMU-CS-00-104, Carnegie Mellon University, School of Computer Science, January 2000.

[122] Crary, Karl, Robert Harper, and Derek Dreyer. A type system for higher-order modules. In *ACM Symposium on Principles of Programming Languages (POPL), Portland, Oregon*, 2002.

[123] Crary, Karl, Robert Harper, and Sidd Puri. What is a recursive module? In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 50–63, May 1999.

[124] Crary, Karl, Stephanie Weirich, and J. Gregory Morrisett. Intensional polymorphism in type-erasure semantics. In *International Conference on Functional Programming (ICFP), Baltimore, Maryland, USA*, pages 301–312, 1998.

[125] Crole, Roy. *Categories for Types.* Cambridge University Press, 1994.

[126] Curien, Pierre-Louis and Giorgio Ghelli. Subtyping + extensionality: Confluence of $\beta\eta$-reductions in $F_{\leq}$. In T. Ito and A. R. Meyer, editors, *Theoretical Aspects of Computer Software (Sendai, Japan)*, number 526 in Lecture Notes in Computer Science, pages 731–749. Springer-Verlag, September 1991.

[127] Curien, Pierre-Louis and Giorgio Ghelli. Coherence of subsumption: Minimum typing and type-checking in $\mathrm{F}_{\leq}$. *Mathematical Structures in Computer Science*, 2:55–91, 1992. Also in Carl A. Gunter and John C. Mitchell, editors, *Theoretical Aspects of Object-Oriented Programming: Types, Semantics, and Language Design* (MIT Press, 1994).

[128] Curry, Haskell B. and Robert Feys. *Combinatory Logic*, volume 1. North Holland, 1958. Second edition, 1968.

[129] Damas, Luis and Robin Milner. Principal type schemes for functional programs. In *ACM Symposium on Principles of Programming Languages (POPL), Albuquerque, New Mexico*, pages 207–212, 1982.

[130] Danvy, Olivier. Type-directed partial evaluation. In John Hatcliff, Torben Æ. Mogensen, and Peter Thiemann, editors, *Partial Evaluation – Practice and Theory; Proceedings of*

the 1998 DIKU Summer School, number 1706 in Lecture Notes in Computer Science, pages 367–411, Copenhagen, Denmark, July 1998. Springer-Verlag.

[131] Davey, Brian A. and Hilary A. Priestley. *Introduction to Lattices and Order.* Cambridge University Press, 1990.

[132] Davies, Rowan. A refinement-type checker for Standard ML. In *International Conference on Algebraic Methodology and Software Technology*, volume 1349 of *Lecture Notes in Computer Science.* Springer-Verlag, 1997.

[133] Davies, Rowan and Frank Pfenning. A modal analysis of staged computation. In *ACM Symposium on Principles of Programming Languages (POPL), St. Petersburg Beach, Florida*, pages 258–270, 1996.

[134] de Bruijn, Nicolas G. Lambda-calculus notation with nameless dummies: a tool for automatic formula manipulation with application to the Church-Rosser theorem. *Indag. Math.*, 34(5):381–392, 1972.

[135] de Bruijn, Nicolas G. A survey of the project AUTOMATH. In J. P. Seldin and J. R. Hindley, editors, *To H. B. Curry: Essays in Combinatory Logic, Lambda Calculus, and Formalism*, pages 589–606. Academic Press, 1980.

[136] De Millo, Richard A., Richard J. Lipton, and Alan J. Perlis. Social processes and proofs of theorems and programs. *Communications of the ACM*, 22(5):271–280, May 1979. An earlier version appeared in *ACM Symposium on Principles of Programming Languages (POPL), Los Angeles, California*, 1977 pp. 206–214.

[137] Detlefs, David L., K. Rustan M. Leino, Greg Nelson, and James B. Saxe. Extended static checking. Technical Report 159, Compaq Systems Research Center (SRC), 1998. Also see `http://research.compaq.com/SRC/esc/overview.html`.

[138] Donahue, James and Alan Demers. Data types are values. *ACM Transactions on Programming Languages and Systems*, 7(3):426–445, July 1985.

[139] Dowek, Gilles, Thérèse Hardin, Claude Kirchner, and Frank Pfenning. Unification via explicit substitutions: The case of higher-order patterns. In M. Maher, editor, *Proceedings of the Joint International Conference and Symposium on Logic Programming*, pages 259–273, Bonn, Germany, September 1996. MIT Press.

[140] Drossopoulou, Sophia, Susan Eisenbach, and Sarfraz Khurshid. Is the Java type system sound? *Theory and Practice of Object Systems*, 5(1):3–24, 1999. Summary in European Conference on Object-Oriented Programming (ECOOP), 1997.

[141] Duggan, Dominic and Adriana Compagnoni. Subtyping for object type constructors. In *Workshop on Foundations of Object-Oriented Languages (FOOL), informal proceedings*, January 1999.

[142] Dybvig, R. Kent. *The Scheme Programming Language.* Prentice-Hall, Inc., Englewood Cliffs, New Jersey, second edition, 1996. Available electronically at `http://www.scheme.com/tspl2d/`.

[143] Eidorff, Peter, Fritz Henglein, Christian Mossin, Henning Niss, Morten Heine B. Sørensen, and Mads Tofte. AnnoDomini in practice: A type-theoretic approach to the Year 2000 problem. In Jean-Yves Girard, editor, *Proc. Symposium on Typed Lambda Calculus and Applications (TLCA)*, volume 1581 of *Lecture Notes in Computer Science*, pages 6–13, L'Aquila, Italy, April 1999. Springer-Verlag.

[144] Eifrig, Jonathan, Scott Smith, and Valery Trifonov. Type inference for recursively constrained types and its application to OOP. In *Proceedings of the 1995 Mathematical Foundations of Programming Semantics Conference*, volume 1 of *Electronic Notes in Theoretical Computer Science*. Elsevier, 1995.

[145] Feinberg, Neal, Sonya E. Keene, Robert O. Mathews, and P. Tucker Withington. *The Dylan Programming Book*. Addison-Wesley Longman, Reading, Mass., 1997.

[146] Felleisen, Matthias and Daniel P. Friedman. *A Little Java, A Few Patterns*. MIT Press, Cambridge, Massachusetts, 1998.

[147] Felty, Amy, Elsa Gunter, John Hannan, Dale Miller, Gopalan Nadathur, and Andre Scedrov. Lambda prolog: An extended logic programming language. In E. Lusk and R. Overbeek, editors, *Proceedings on the 9th International Conference on Automated Deduction*, volume 310 of *LNCS*, pages 754–755, Berlin, May 1988. Springer.

[148] Filinski, Andrzej. A semantic account of type-directed partial evaluation. In Gopalan Nadathur, editor, *Proceedings of the International Conference on Principles and Practice of Declarative Programming*, number 1702 in Lecture Notes in Computer Science, pages 378–395, Paris, France, September 1999. Springer-Verlag. Extended version available as technical report BRICS RS-99-17.

[149] Filinski, Andrzej. Normalization by evaluation for the computational lambda-calculus. In Samson Abramsky, editor, *Typed Lambda Calculi and Applications, 5th International Conference, TLCA 2001*, number 2044 in Lecture Notes in Computer Science, pages 151–165, Kraków, Poland, May 2001. Springer-Verlag.

[150] Fisher, Kathleen. Classes = objects + data abstraction. In Kim Bruce and Giuseppe Longo, editors, *Workshop on Foundations of Object-Oriented Languages (FOOL), informal proceedings*, July 1996. Invited talk. Also available as Stanford University Technical Note STAN-CS-TN-96-31.

[151] Fisher, Kathleen. *Type Systems for object-oriented programming languages*. PhD thesis, Stanford University, 1996. STAN-CS-TR-98-1602.

[152] Fisher, Kathleen, Furio Honsell, and John C. Mitchell. A lambda calculus of objects and method specialization. *Nordic J. Computing (*formerly *BIT)*, 1:3–37, 1994. Summary in *Proc. IEEE Symp. on Logic in Computer Science,* 1993, 26–38.

[153] Fisher, Kathleen and John Mitchell. The development of type systems for object-oriented languages. *Theory and Practice of Object Systems*, 1(3):189–220, 1996.

[154] Fisher, Kathleen and John C. Mitchell. On the relationship between classes, objects, and data abstraction. *Theory and Practice of Object Systems*, 4(1):3–25, 1998.

[155] Fisher, Kathleen and John H. Reppy. The design of a class mechanism for Moby. In *SIGPLAN Conference on Programming Language Design and Implementation (PDLI)*, pages 37–49, 1999.

[156] Flanagan, Cormac and Matthias Felleisen. Componential set-based analysis. *ACM SIGPLAN Notices*, 32(5):235–248, May 1997.

[157] Flatt, Matthew and Matthias Felleisen. Units: Cool modules for HOT languages. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), Montreal, Canada*, pages 236–248, 1998.

[158] Flatt, Matthew, Shriram Krishnamurthi, and Matthias Felleisen. Classes and mixins.

In *ACM Symposium on Principles of Programming Languages (POPL), San Diego, California*, January 1998.

[159] Flatt, Matthew, Shriram Krishnamurthi, and Matthias Felleisen. A programmer's reduction semantics for classes and mixins. Technical Report TR97-293, Computer Science Department, Rice University, February 1998. Corrected June, 1999.

[160] Freeman, Tim and Frank Pfenning. Refinement types for ML. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), Toronto, Ontario*, June 1991.

[161] Frege, Gottlob. *Begriffschrift, eine der arithmetischen nachgebildete Formelsprache des reinen Denkens.* Halle: L. Nebert, 1879. Available in several translations.

[162] Friedman, Daniel P. and Matthias Felleisen. *The Little Schemer.* MIT Press, 1996.

[163] Friedman, Daniel P., Mitchell Wand, and Christopher T. Haynes. *Essentials of Programming Languages.* McGraw-Hill Book Co., New York, N.Y., second edition, 2001.

[164] Friedman, Harvey. Equality between functionals. In Rohit Parikh, editor, *Logic Colloquium*, volume 453 of *Lecture Notes in Mathematics*, pages 22–37, Berlin, 1975. Springer-Verlag.

[165] Gallier, Jean. On Girard's "Candidats de reductibilité". In Piergiorgio Odifreddi, editor, *Logic and Computer Science*, number 31 in APIC Studies in Data Processing, pages 123–203. Academic Press, 1990.

[166] Gallier, Jean. Constructive logics. Part I: A tutorial on proof systems and typed λ-calculi. *Theoretical Computer Science*, 110(2):249–339, March 1993.

[167] Gandy, Robin O. The simple theory of types. In *Logic Colloquium 76*, volume 87 of *Studies in Logic and the Foundations of Mathematics*, pages 173–181. North Holland, 1976.

[168] Gapeyev, Vladimir, Michael Levin, and Benjamin Pierce. Recursive subtyping revealed. In *International Conference on Functional Programming (ICFP), Montreal, Canada*, 2000. To appear in *Journal of Functional Programming.*

[169] Garrigue, Jacques and Hassan Aït-Kaci. The typed polymorphic label-selective lambda-calculus. In *ACM Symposium on Principles of Programming Languages (POPL), Portland, Oregon*, pages 35–47, 1994.

[170] Garrigue, Jacques and Didier Rémy. Extending ML with semi-explicit polymorphism. In Martín Abadi and Takayasu Ito, editors, *International Symposium on Theoretical Aspects of Computer Software (TACS), Sendai, Japan*, pages 20–46. Springer-Verlag, September 1997.

[171] Ghelli, Giorgio. *Proof Theoretic Studies about a Minimal Type System Integrating Inclusion and Parametric Polymorphism.* PhD thesis, Università di Pisa, March 1990. Technical report TD-6/90, Dipartimento di Informatica, Università di Pisa.

[172] Ghelli, Giorgio. Recursive types are not conservative over $F_{\leq}$. In M. Bezen and J.F. Groote, editors, *Typed Lambda Calculi and Applications (TLCA), Utrecht, The Netherlands*, number 664 in Lecture Notes in Computer Science, pages 146–162, Berlin, March 1993. Springer-Verlag.

[173] Ghelli, Giorgio. Divergence of $F_{\leq}$ type checking. *Theoretical Computer Science*, 139 (1,2):131–162, 1995.

[174] Ghelli, Giorgio. Termination of system F-bounded: A complete proof. *Information and Computation*, 139(1):39–56, 1997.

[175] Ghelli, Giorgio and Benjamin Pierce. Bounded existentials and minimal typing. *Theoretical Computer Science*, 193:75–96, 1998.

[176] Gifford, David, Pierre Jouvelot, John Lucassen, and Mark Sheldon. FX-87 Reference Manual. Technical Report MIT/LCS/TR-407, Massachusetts Institute of Technology, Laboratory for Computer Science, September 1987.

[177] Girard, Jean-Yves. *Interprétation fonctionnelle et élimination des coupures de l'arithmétique d'ordre supérieur.* Thèse d'état, Université Paris VII, 1972. Summary in *Proceedings of the Second Scandinavian Logic Symposium* (J.E. Fenstad, editor), North-Holland, 1971 (pp. 63–92).

[178] Girard, Jean-Yves. Linear logic. *Theoretical Computer Science*, 50:1–102, 1987.

[179] Girard, Jean-Yves, Yves Lafont, and Paul Taylor. *Proofs and Types*, volume 7 of *Cambridge Tracts in Theoretical Computer Science.* Cambridge University Press, 1989.

[180] Glew, Neal. Type dispatch for named hierarchical types. In *International Conference on Functional Programming (ICFP), Paris, France*, pages 172–182, 1999.

[181] Gordon, Andrew. A tutorial on co-induction and functional programming. In *Functional Programming, Glasgow 1994*, pages 78–95. Springer Workshops in Computing, 1995.

[182] Gordon, Michael J. Adding eval to ML. Manuscript, circa 1980.

[183] Gordon, Michael J., Robin Milner, and Christopher P. Wadsworth. *Edinburgh LCF.* Springer-Verlag LNCS 78, 1979.

[184] Goto, Eiichi. Monocopy and associative algorithms in extended Lisp. Technical Report TR 74-03, University of Tokyo, May 1974.

[185] Goubault-Larrecq, Jean and Ian Mackie. *Proof Theory and Automated Deduction (Applied Logic Series, V. 6).* Kluwer, 1997.

[186] Grattan-Guinness, Ivor. *The Search for Mathematical Roots, 1870–1940: Logics, Set Theories and the Foundations of Mathematics from Cantor through Russell to Gödel.* Princeton University Press, 2001.

[187] Gries, David, editor. *Programming Methodology.* Springer-Verlag, New York, 1978.

[188] Gunter, Carl A. *Semantics of Programming Languages: Structures and Techniques.* MIT Press, 1992.

[189] Gunter, Carl A. and John C. Mitchell. *Theoretical Aspects of Object-Oriented Programming: Types, Semantics, and Language Design.* MIT Press, 1994.

[190] Hall, Cordelia V., Kevin Hammond, Simon L. Peyton Jones, and Philip L. Wadler. Type classes in Haskell. *ACM Transactions on Programming Languages and Systems*, 18(2): 109–138, March 1996.

[191] Halmos, Paul R. *Naive Set Theory.* Springer, New York, 1987.

[192] Harper, Robert. A simplified account of polymorphic references. *Information Processing Letters*, 51(4):201–206, August 1994. See also Harper [193].

[193] Harper, Robert. A note on: "A simplified account of polymorphic references" [Inform. Process. Lett. **51** (1994), no. 4, 201–206; MR 95f:68142]. *Information Processing Letters*, 57(1):15–16, January 1996. See Harper [192].

[194] Harper, Robert, Bruce Duba, and David MacQueen. First-class continuations in ML. *Journal of Functional Programming*, 3(4), October 1993. Short version in POPL '91.

[195] Harper, Robert, Furio Honsell, and Gordon Plotkin. A framework for defining logics. *Journal of the ACM*, 40(1):143–184, 1992. Summary in LICS'87.

[196] Harper, Robert and Mark Lillibridge. A type-theoretic approach to higher-order modules with sharing. In *ACM Symposium on Principles of Programming Languages (POPL), Portland, Oregon*, pages 123–137, January 1994.

[197] Harper, Robert, John C. Mitchell, and Eugenio Moggi. Higher-order modules and the phase distinction. In *ACM Symposium on Principles of Programming Languages (POPL), San Francisco, California*, pages 341–354, January 1990.

[198] Harper, Robert and Greg Morrisett. Compiling polymorphism using intensional type analysis. In *ACM Symposium on Principles of Programming Languages (POPL), San Francisco, California*, pages 130–141, 1995.

[199] Harper, Robert and Christopher Stone. A type-theoretic interpretation of Standard ML. In Gordon Plotkin, Colin Stirling, and Mads Tofte, editors, *Proof, Language and Interaction: Essays in Honour of Robin Milner*. MIT Press, 2000.

[200] Hasegawa, Ryu. Parametricity of extensionally collapsed term models of polymorphism and their categorical properties. In Takayasu Ito and Albert Meyer, editors, *Theoretical Aspects of Computer Software (TACS), Sendai, Japan*, 1991.

[201] Hayashi, Susumu. Singleton, union and intersection types for program extraction. In T. Ito and A. R. Meyer, editors, *Theoretical Aspects of Computer Software (Sendai, Japan)*, number 526 in Lecture Notes in Computer Science, pages 701–730. Springer-Verlag, September 1991. Full version in *Information and Computation*, 109(1/2):174–210, 1994.

[202] Henglein, Fritz. Type inference with polymorphic recursion. *ACM Transactions on Programming Languages and Systems*, 15(2):253–289, 1993.

[203] Henglein, Fritz. Dynamic typing: syntax and proof theory. *Science of Computer Programming*, 22(3):197–230, June 1994. Selected papers of the Fourth European Symposium on Programming (Rennes, 1992).

[204] Henglein, Fritz and Harry G. Mairson. The complexity of type inference for higher-order typed lambda-calculi. In *ACM Symposium on Principles of Programming Languages (POPL), Orlando, Florida*, pages 119–130, January 1991.

[205] Hennessy, Matthew. *The Semantics of Programming Languages: An Elementary Introduction Using Operational Semantics*. John Wiley and Sons, 1990. Currently out of print.

[206] Hennessy, Matthew and James Riely. Resource access control in systems of mobile agents. In Uwe Nestmann and Benjamin C. Pierce, editors, *HLCL '98: High-Level Concurrent Languages (Nice, France, September 12, 1998)*, volume 16.3 of *ENTCS*, pages 3–17. Elsevier Science Publishers, 1998. Full version available as CogSci Report 2/98, University of Sussex, Brighton.

[207] Hindley, J. Roger. The principal type-scheme of an object in combinatory logic. *Transactions of the American Mathematical Society*, 146:29–60, 1969.

[208] Hindley, J. Roger. Types with intersection, an introduction. *Formal Aspects of*

Computing, 4:470–486, 1992.

[209] Hindley, J. Roger. *Basic Simple Type Theory*, volume 42 of *Cambridge Tracts in Theoretical Computer Science*. Cambridge University Press, Cambridge, 1997.

[210] Hindley, J. Roger and Jonathan P. Seldin. *Introduction to Combinators and λ-Calculus*, volume 1 of *London Mathematical Society Student Texts*. Cambridge University Press, 1986.

[211] Hoang, My, John Mitchell, and Ramesh Viswanathan. Standard ML-NJ weak polymorphism and imperative constructs. In *Proceedings, Eighth Annual IEEE Symposium on Logic in Computer Science*, pages 15–25. IEEE Computer Society Press, 1993.

[212] Hodas, J. S. Lolli: An extension of λProlog with linear context management. In D. Miller, editor, *Workshop on the λProlog Programming Language*, pages 159–168, Philadelphia, Pennsylvania, August 1992.

[213] Hofmann, Martin. Syntax and semantics of dependent types. In *Semantics and Logic of Computation*. Cambridge University Press, 1997.

[214] Hofmann, Martin and Benjamin Pierce. Positive subtyping. In *ACM Symposium on Principles of Programming Languages (POPL), San Francisco, California*, pages 186–197, January 1995. Full version in *Information and Computation*, volume 126, number 1, April 1996. Also available as University of Edinburgh technical report ECS-LFCS-94-303, September 1994.

[215] Hofmann, Martin and Benjamin Pierce. A unifying type-theoretic framework for objects. *Journal of Functional Programming*, 5(4):593–635, October 1995. Previous versions appeared in the Symposium on Theoretical Aspects of Computer Science, 1994, (pages 251–262) and, under the title "An Abstract View of Objects and Subtyping (Preliminary Report)," as University of Edinburgh, LFCS technical report ECS-LFCS-92-226, 1992.

[216] Hofmann, Martin and Benjamin C. Pierce. Type destructors. In Didier Rémy, editor, *Informal proceedings of the Fourth International Workshop on Foundations of Object-Oriented Languages (FOOL)*, January 1998. Full version in *Information and Computation*, 172(1), Jan 2002, pp. 29–62.

[217] Hook, J.G. Understanding Russell – a first attempt. In *Proc. Int. Symp. on Semantics of Data Types, Sophia-Antipolis (France), Springer LNCS 173*, pages 69–85. Springer-Verlag, 1984.

[218] Hopcroft, John E. and Jeffrey D. Ullman. *Introduction to Automata Theory, Languages, and Computation*. Addison-Wesley, 1979.

[219] Hosoya, Haruo and Benjamin Pierce. Regular expression pattern matching. In *ACM Symposium on Principles of Programming Languages (POPL), London, England*, 2001.

[220] Hosoya, Haruo and Benjamin C. Pierce. How good is local type inference? Technical Report MS-CIS-99-17, University of Pennsylvania, June 1999.

[221] Hosoya, Haruo and Benjamin C. Pierce. XDuce: A typed XML processing language (preliminary report). In *International Workshop on the Web and Databases (WebDB)*, May 2000.

[222] Hosoya, Haruo, Jérôme Vouillon, and Benjamin C. Pierce. Regular expression types

for XML. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 27(1): 46–90, January 2005. Short version in ICFP 2000, 35(9):11–22, September 2000.

[223] Howard, William A. Hereditarily majorizable functionals of finite type. In Anne Sjerp Troelstra, editor, *Metamathematical Investigation of Intuitionistic Arithmetic and Analysis*, volume 344 of *Lecture Notes in Mathematics*, pages 454–461. Springer-Verlag, Berlin, 1973. Appendix.

[224] Howard, William A. The formulas-as-types notion of construction. In J. P. Seldin and J. R. Hindley, editors, *To H. B. Curry: Essays on Combinatory Logic, Lambda Calculus, and Formalism*, pages 479–490. Academic Press, New York, 1980. Reprint of 1969 article.

[225] Howe, Douglas. *Automating Reasoning in an Implementation of Constructive Type Theory*. PhD thesis, Cornell University, 1988.

[226] Hudak, Paul, S. Peyton Jones, P. Wadler, B. Boutel, J. Fairbairn, J. Fasel, M. M. Guzman, K. Hammond, J. Hughes, T. Johnsson, D. Kieburtz, R. Nikhil, W. Partain, and J. Peterson. Report on the programming language Haskell, version 1.2. *SIGPLAN Notices*, 27(5), May 1992.

[227] Huet, Gérard. A unification algorithm for typed λ-calculus. *Theoretical Computer Science*, 1:27–57, 1975.

[228] Huet, Gérard. *Résolution d'equations dans les langages d'ordre 1,2,...,ω*. Thèse de Doctorat d'Etat, Université de Paris 7 (France), 1976.

[229] Huet, Gérard, editor. *Logical Foundations of Functional Programming*. University of Texas at Austin Year of Programming Series. Addison-Wesley, 1990.

[230] Hyland, J. Martin E. and C.-H. Luke Ong. On full abstraction for PCF: I, II, and III. *Information and Computation*, 163(2):285–408, December 2000.

[231] Igarashi, Atsushi, Benjamin Pierce, and Philip Wadler. Featherweight Java: A minimal core calculus for Java and GJ. In *ACM Symposium on Object Oriented Programming: Systems, Languages, and Applications (OOPSLA)*, October 1999. Full version in ACM Transactions on Programming Languages and Systems (TOPLAS), 23(3), May 2001, pp. 396–450.

[232] Igarashi, Atsushi and Benjamin C. Pierce. On inner classes. In *European Conference on Object-Oriented Programming (ECOOP)*, 2000. Also in informal proceedings of the Seventh International Workshop on Foundations of Object-Oriented Languages (FOOL). In *Information and Computation*, 177(1), Aug 2002, pp. 56–89.

[233] Igarashi, Atsushi, Benjamin C. Pierce, and Philip Wadler. A recipe for raw types. In *Workshop on Foundations of Object-Oriented Languages (FOOL)*, 2001.

[234] Ishtiaq, Samin and Peter O'Hearn. Bi as an assertion language for mutable data structures. In *ACM Symposium on Principles of Programming Languages (POPL), London, England*, 2001.

[235] Jacobs, Bart. *Categorical Logic and Type Theory*. Number 141 in Studies in Logic and the Foundations of Mathematics. North Holland, Elsevier, 1999.

[236] Jagannathan, Suresh and Andrew Wright. Effective flow analysis for avoiding run-time checks. In *Proceedings of the Second International Static Analysis Symposium*, volume 983 of *LNCS*, pages 207–224. Springer-Verlag, 1995.

[237] Jay, C. Barry and Milan Sekanina. Shape checking of array programs. In *Computing: The Australasian Theory Seminar (Proceedings)*, volume 19 of *Australian Computer Science Communications*, pages 113–121, 1997.

[238] Jim, Trevor. Rank-2 type systems and recursive definitions. Technical Report MIT/LCS/TM-531, Massachusetts Institute of Technology, Laboratory for Computer Science, November 1995.

[239] Jim, Trevor. What are principal typings and what are they good for? In *ACM Symposium on Principles of Programming Languages (POPL), St. Petersburg Beach, Florida*, pages 42–53. ACM, 1996.

[240] Jim, Trevor and Jens Palsberg. Type inference in systems of recursive types with subtyping. Manuscript, 1999.

[241] Jones, Mark P. ML typing, explicit polymorphism, and qualified types, 1994.

[242] Jones, Mark P. *Qualified Types: Theory and Practice*. Cambridge University Press, 1994.

[243] Jones, Richard and Rafael D. Lins. *Garbage Collection: Algorithms for Automatic Dynamic Memory Management*. Wiley, 1996.

[244] Jouvelot, Pierre and David Gifford. Algebraic reconstruction of types and effects. In *ACM Symposium on Principles of Programming Languages (POPL), Orlando, Florida*, pages 303–310, January 1991.

[245] Jutting, L.S. van Benthem, James McKinna, and Robert Pollack. Checking algorithms for Pure Type Systems. In Henk Barendregt and Tobias Nipkow, editors, *Proceedings of the International Workshop on Types for Proofs and Programs*, pages 19–61, Nijmegen, The Netherlands, May 1994. Springer-Verlag LNCS 806.

[246] Kaes, Stefan. Parametric overloading in polymorphic programming languages. In H. Ganzinger, editor, *Proceedings of the European Symposium on Programming*, volume 300 of *Lecture Notes in Computer Science*, pages 131–144. Springer-Verlag, 1988.

[247] Kahn, Gilles. Natural semantics. In *Proceedings of the Symposium on Theoretical Aspects of Computer Science (STACS)*, volume 247 of *Lecture Notes in Computer Science*, pages 22–39. Springer-Verlag, 1987.

[248] Kamin, Samuel N. Inheritance in Smalltalk-80: A denotational definition. In *ACM Symposium on Principles of Programming Languages (POPL), San Diego, California*, pages 80–87, January 1988.

[249] Kamin, Samuel N. and Uday S. Reddy. Two semantic models of object-oriented languages. In Carl A. Gunter and John C. Mitchell, editors, *Theoretical Aspects of Object-Oriented Programming: Types, Semantics, and Language Design*, pages 464–495. MIT Press, 1994.

[250] Katiyar, Dinesh, David Luckham, and John Mitchell. A type system for prototyping languages. In *ACM Symposium on Principles of Programming Languages (POPL), Portland, Oregon*, pages 138–150, January 1994.

[251] Katiyar, Dinesh and Sriram Sankar. Completely bounded quantification is decidable. In *Proceedings of the ACM SIGPLAN Workshop on ML and its Applications*, June 1992.

[252] Kelsey, Richard, William Clinger, and Jonathan Rees. Revised^5 report on the algorithmic language Scheme. *Higher-Order and Symbolic Computation*, 11(1):7–105,

1998. Also appears in ACM SIGPLAN Notices 33(9), September 1998.

[253] Kennedy, Andrew. Dimension types. In Donald Sannella, editor, *Programming Languages and Systems—ESOP'94, 5th European Symposium on Programming*, volume 788 of *Lecture Notes in Computer Science*, pages 348–362, Edinburgh, U.K., 11–13 April 1994. Springer.

[254] Kernighan, Brian W. and Dennis M. Ritchie. *The C Programming Language.* Prentice Hall, Englewood Cliffs, second edition, 1988.

[255] Kfoury, A. J. and J. B. Wells. A direct algorithm for type inference in the rank-2 fragment of the second-order λ-calculus. In *Proceedings of the 1994 ACM conference on LISP and functional programming*, pages 196–207, New York, NY, USA, 1994. ACM.

[256] Kfoury, Assaf J., Harry Mairson, Franklyn Turbak, and Joe B. Wells. Relating typability and expressiveness in finite-rank intersection type systems. In *International Conference on Functional Programming (ICFP), Paris, France*, volume 34.9 of *ACM Sigplan Notices*, pages 90–101, N.Y., September 27–29 1999. ACM Press.

[257] Kfoury, Assaf J. and Jerzy Tiuryn. Type reconstruction in finite-rank fragments of the polymorphic λ-calculus. In *Fifth Annual IEEE Symposium on Logic in Computer Science*, pages 2–11, Philadelphia, PA, June 1990. Full version in *Information and Computation*, 98(2), 228–257, 1992.

[258] Kfoury, Assaf J., Jerzy Tiuryn, and Pawel Urzyczyn. ML typability is DEXPTIME-complete. In *Proc. 15th Colloq. on Trees in Algebra and Programming*, pages 206–220. Springer LNCS 431, 1990.

[259] Kfoury, Assaf J., Jerzy Tiuryn, and Pawel Urzyczyn. Type reconstruction in the presence of polymorphic recursion. *ACM Transactions on Programming Languages and Systems*, 15(2):290–311, April 1993.

[260] Kfoury, Assaf J., Jerzy Tiuryn, and Pawel Urzyczyn. The undecidability of the semi-unification problem. *Information and Computation*, 102(1):83–101, January 1993. Summary in *STOC 1990*.

[261] Kfoury, Assaf J., Jerzy Tiuryn, and Pawel Urzyczyn. An analysis of ML typability. *Journal of the ACM*, 41(2):368–398, March 1994.

[262] Kiczales, Gregor, Jim des Rivières, and Daniel G. Bobrow. *The Art of the Metaobject Protocol.* MIT Press, Cambridge, MA, 1991.

[263] Kirchner, Claude and Jean-Pierre Jouannaud. Solving equations in abstract algebras: a rule-based survey of unification. Research Report 561, Université de Paris Sud, Orsay, France, April 1990.

[264] Klop, Jan W. *Combinatory Reduction Systems.* Mathematical Centre Tracts 127. Mathematisch Centrum, Amsterdam, 1980.

[265] Kobayashi, Naoki, Benjamin C. Pierce, and David N. Turner. Linearity and the pi-calculus. In *ACM Symposium on Principles of Programming Languages (POPL), St. Petersburg Beach, Florida*, 1996. Full version in *ACM Transactions on Programming Languages and Systems*, 21(5), pp. 914–947, September 1999.

[266] Kozen, Dexter, Jens Palsberg, and Michael I. Schwartzbach. Efficient recursive subtyping. In *ACM Symposium on Principles of Programming Languages (POPL), Charleston, South Carolina*, pages 419–428, 1993.

[267] Laan, Twan Dismas Laurens. *The Evolution of Type Theory in Logic and Mathematics.* PhD thesis, Techn. Univ. Eindhoven, 1997.

[268] Landin, Peter J. The mechanical evaluation of expressions. *Computer Journal*, 6:308-320, January 1964.

[269] Landin, Peter J. A correspondence between ALGOL 60 and Church's lambda-notation: Parts I and II. *Communications of the ACM*, 8(2,3):89-101, 158-165, February and March 1965.

[270] Landin, Peter J. The next 700 programming languages. *Communications of the ACM*, 9 (3):157-166, March 1966.

[271] Lassez, Jean-Louis and Gordin Plotkin, editors. *Computational Logic, Essays in Honor of Alan Robinson.* MIT Press, 1991.

[272] Läufer, Konstantin. *Polymorphic Type Inference and Abstract Data Types.* PhD thesis, New York University, 1992.

[273] Läufer, Konstantin and Martin Odersky. Polymorphic type inference and abstract data types. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 16(5):1411-1430, September 1994. Summary in *Phoenix Seminar and Workshop on Declarative Programming*, Nov. 1991.

[274] League, Christopher, Zhong Shao, and Valery Trifonov. Representing Java classes in a typed intermediate language. In *International Conference on Functional Programming (ICFP), Paris, France*, September 1999.

[275] League, Christopher, Valery Trifonov, and Zhong Shao. Type-preserving compilation of Featherweight Java. In *Foundations of Object-Oriented Languages (FOOL8)*, London, January 2001.

[276] Lee, Oukseh and Kwangkeun Yi. Proofs about a folklore let-polymorphic type inference algorithm. *ACM Transactions on Programming Languages and Systems*, 20(4):707-723, July 1998.

[277] Leivant, Daniel. Polymorphic type inference. In *Proceedings of the 10th Annual ACM Symposium on Principles of Programming Languages.* ACM, 1983.

[278] Lemmon, E. John, Carew A. Meredith, David Meredith, Arthur N. Prior, and Ivo Thomas. Calculi of pure strict implication, 1957. Mimeographed version, 1957; published in Davis, Hockney, and Wilson, editors, *Philosophical Logic*, D. Reidel Co., Netherlands, 1969, pp. 215-250.

[279] Leroy, Xavier. Manifest types, modules and separate compilation. In *ACM Symposium on Principles of Programming Languages (POPL), Portland, Oregon*, pages 109-122, Portland, OR, January 1994.

[280] Leroy, Xavier. The Objective Caml system: Documentation and user's manual, 2000. With Damien Doligez, Jacques Garrigue, Didier Rémy, and Jérôme Vouillon. Available from `http://caml.inria.fr`.

[281] Leroy, Xavier and Michel Mauny. Dynamics in ML. In John Hughes, editor, *ACM Symposium on Functional Programming Languages and Computer Architecture (FPCA) 1991*, volume 523 of *Lecture Notes in Computer Science*, pages 406-426. Springer-Verlag, 1991.

[282] Leroy, Xavier and François Pessaux. Type-based analysis of uncaught exceptions. *ACM*

Transactions on Programming Languages and Systems, 22(2):340–377, March 2000. Summary in *ACM Symposium on Principles of Programming Languages (POPL), San Antonio, Texas*, 1999.

[283] Leroy, Xavier and François Rouaix. Security properties of typed applets. In *ACM Symposium on Principles of Programming Languages (POPL), San Diego, California*, pages 391–403, January 1998.

[284] Leroy, Xavier and Pierre Weis. Polymorphic type inference and assignment. In *ACM Symposium on Principles of Programming Languages (POPL), Orlando, Florida*, pages 291–302, 1991.

[285] Lescanne, Pierre and Jocelyn Rouyer-Degli. Explicit substitutions with de Bruijn's levels. In J. Hsiang, editor, *Proceedings of the 6th Conference on Rewriting Techniques and Applications (RTA), Kaiserslautern (Germany)*, volume 914, pages 294–308, 1995.

[286] Levin, Michael Y. and Benjamin C. Pierce. Tinkertype: A language for playing with formal systems. *Journal of Functional Programming*, 13(2):295–316, March 2003. A preliminary version appeared as an invited talk at the *Logical Frameworks and Metalanguages Workshop (LFM)*, June 2000.

[287] Lillibridge, Mark. *Translucent Sums: A Foundation for Higher-Order Module Systems.* PhD thesis, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA, May 1997.

[288] Liskov, Barbara, Russell Atkinson, Toby Bloom, Elliott Moss, J. Craig Schaffert, Robert Scheifler, and Alan Snyder. *CLU Reference Manual.* Springer-Verlag, 1981.

[289] Liskov, Barbara, Alan Snyder, Russell Atkinson, and J. Craig Schaffert. Abstraction mechanisms in CLU. *Communications of the ACM*, 20(8):564–576, August 1977. Also in S. Zdonik and D. Maier, eds., *Readings in Object-Oriented Database Systems.*

[290] Luo, Zhaohui. *Computation and Reasoning: A Type Theory for Computer Science.* Number 11 in International Series of Monographs on Computer Science. Oxford University Press, 1994.

[291] Luo, Zhaohui and Robert Pollack. The LEGO proof development system: A user's manual. Technical Report ECS-LFCS-92-211, University of Edinburgh, May 1992.

[292] Ma, QingMing. Parametricity as subtyping. In *ACM Symposium on Principles of Programming Languages (POPL), Albuquerque, New Mexico*, January 1992.

[293] Mackie, Ian. Lilac: A functional programming language based on linear logic. *Journal of Functional Programming*, 4(4):395–433, October 1994.

[294] MacQueen, David. Using dependent types to express modular structure. In *ACM Symposium on Principles of Programming Languages (POPL), St. Petersburg Beach, Florida*, pages 277–286, January 1986.

[295] MacQueen, David, Gordon Plotkin, and Ravi Sethi. An ideal model for recursive polymorphic types. *Information and Control*, 71:95–130, 1986.

[296] MacQueen, David B. Using dependent types to express modular structure. In *ACM Symposium on Principles of Programming Languages (POPL), St. Petersburg Beach, Florida*, 1986.

[297] Magnusson, Lena and Bengt Nordström. The ALF proof editor and its proof engine. In Henk Barendregt and Tobias Nipkow, editors, *Types for Proofs and Programs*, pages

213–237. Springer-Verlag LNCS 806, 1994.

[298] Mairson, Harry G. Deciding ML typability is complete for deterministic exponential time. In *ACM Symposium on Principles of Programming Languages (POPL), San Francisco, California*, pages 382–401. ACM Press, New York, 1990.

[299] Martin-Löf, Per. An intuitionistic theory of types: predicative part. In H. E. Rose and J. C. Shepherdson, editors, *Logic Colloquium, '73*, pages 73–118. North-Holland, Amsterdam, 1973.

[300] Martin-Löf, Per. Constructive mathematics and computer programming. In *Logic, Methodology and Philosophy of Science, VI.* North Holland, Amsterdam, 1982.

[301] Martin-Löf, Per. *Intuitionistic Type Theory.* Bibliopolis, 1984.

[302] Martini, Simone. Bounded quantifiers have interval models. In *Proceedings of the ACM Conference on Lisp and Functional Programming*, pages 174–183, Snowbird, Utah, July 1988. ACM.

[303] McCarthy, John. History of LISP. In R. L. Wexelblatt, editor, *History of Programming Languages*, pages 173–197. Academic Press, New York, 1981.

[304] McCarthy, John, S. R. Russell, D. Edwards, et al. *LISP Programmer's Manual.* Massachusetts Institute of Technology, A.I. Lab., Cambridge, Massachusetts, November 1959. Handwritten Draft + Machine Typed.

[305] McKinna, James and Robert Pollack. Pure Type Systems formalized. In M. Bezem and J. F. Groote, editors, *Proceedings of the International Conference on Typed Lambda Calculi and Applications*, pages 289–305. Springer-Verlag LNCS 664, March 1993.

[306] Meertens, Lambert. Incremental polymorphic type checking in B. In *ACM Symposium on Principles of Programming Languages (POPL), Austin, Texas*, 1983.

[307] Milner, Robin. A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, 17:348–375, August 1978.

[308] Milner, Robin. *A Calculus of Communicating Systems*, volume 92 of *Lecture Notes in Computer Science.* Springer-Verlag, 1980.

[309] Milner, Robin. *Communication and Concurrency.* Prentice Hall, 1989.

[310] Milner, Robin. The polyadic π-calculus: a tutorial. Technical Report ECS–LFCS–91–180, Laboratory for Foundations of Computer Science, Department of Computer Science, University of Edinburgh, UK, October 1991. Appeared in F. L. Bauer, W. Brauer, and H. Schwichtenberg, editors, *Proceedings of the International Summer School on Logic and Algebra of Specification*, Marktoberdorf, August 1991. Reprinted in *Logic and Algebra of Specification*, Springer-Verlag, 1993.

[311] Milner, Robin. *Communicating and Mobile Systems: the Pi-Calculus.* Cambridge University Press, 1999.

[312] Milner, Robin, Joachim Parrow, and David Walker. A calculus of mobile processes (Parts I and II). *Information and Computation*, 100:1–77, 1992.

[313] Milner, Robin and Mads Tofte. Co-induction in relational semantics. *Theoretical Computer Science*, 87:209–220, 1991.

[314] Milner, Robin and Mads Tofte. *Commentary on Standard ML.* MIT Press, Cambridge, Massachusetts, 1991.

[315] Milner, Robin, Mads Tofte, and Robert Harper. *The Definition of Standard ML.* MIT

Press, 1990.

[316] Milner, Robin, Mads Tofte, Robert Harper, and David MacQueen. *The Definition of Standard ML (Revised).* MIT Press, 1997.

[317] Mitchell, John C. Coercion and type inference (summary). In *ACM Symposium on Principles of Programming Languages (POPL), Salt Lake City, Utah*, pages 175–185, January 1984.

[318] Mitchell, John C. Type inference and type containment. In *Proc. Int. Symp. on Semantics of Data Types, Sophia-Antipolis (France)*, pages 257–278, Berlin, June 1984. Springer LNCS 173. Full version in G. Huet, editor, *Information and Computation*, vol. 76, no. 2/3, 1988, pp. 211–249. Reprinted in *Logical Foundations of Functional Programming*, Addison-Wesley (1990) 153–194.

[319] Mitchell, John C. Representation independence and data abstraction (preliminary version). In *ACM Symposium on Principles of Programming Languages (POPL), St. Petersburg Beach, Florida*, pages 263–276, 1986.

[320] Mitchell, John C. Toward a typed foundation for method specialization and inheritance. In *ACM Symposium on Principles of Programming Languages (POPL), San Francisco, California*, pages 109–124, January 1990. Also in Carl A. Gunter and John C. Mitchell, editors, *Theoretical Aspects of Object-Oriented Programming: Types, Semantics, and Language Design* (MIT Press, 1994).

[321] Mitchell, John C. Type systems for programming languages. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science, Volume B*, pages 365–458. North-Holland, Amsterdam, 1990.

[322] Mitchell, John C. *Foundations for Programming Languages.* MIT Press, Cambridge, Massachusetts, 1996.

[323] Mitchell, John C. and Robert Harper. The essence of ML. In *ACM Symposium on Principles of Programming Languages (POPL), San Diego, California*, January 1988. Full version in *ACM Transactions on Programming Languages and Systems*, vol. 15, no. 2, 1993, pp. 211–252, under the title "On the type structure of Standard ML".

[324] Mitchell, John C. and Albert R. Meyer. Second-order logical relations (extended abstract). In Rohit Parikh, editor, *Logic of Programs*, volume 193 of *Lecture Notes in Computer Science*, pages 225–236, Berlin, 1985. Springer-Verlag.

[325] Mitchell, John C. and Gordon D. Plotkin. Abstract types have existential type. *ACM Trans. on Programming Languages and Systems*, 10(3):470–502, 1988. Summary in *ACM Symposium on Principles of Programming Languages (POPL), New Orleans, Louisiana*, 1985.

[326] Morris, James H. Lambda calculus models of programming languages. Technical Report MIT-LCS//MIT/LCS/TR-57, Massachusetts Institute of Technology, Laboratory for Computer Science, December 1968.

[327] Morrisett, Greg, Matthias Felleisen, and Robert Harper. Abstract models of memory management. In *Proceedings of the Seventh International Conference on Functional Programming Languages and Computer Architecture (FPCA'95)*, pages 66–77, La Jolla, California, June 25–28, 1995. ACM SIGPLAN/SIGARCH and IFIP WG2.8, ACM Press.

[328] Morrisett, Greg, David Walker, Karl Crary, and Neal Glew. From System F to Typed

Assembly Language. In *ACM Symposium on Principles of Programming Languages (POPL), San Diego, California*, pages 85–97, January 1998.

[329] Mugridge, Warwick B., John Hamer, and John G. Hosking. Multi-methods in a statically-typed programming language. In Pierre America, editor, *ECOOP '91: European Conference on Object-Oriented Programming*, volume 512 of *Lecture Notes in Computer Science*, pages 307–324. Springer-Verlag, 1991.

[330] Mycroft, Alan. Dynamic types in ML. Manuscript, 1983.

[331] Mycroft, Alan. Polymorphic type schemes and recursive definitions. In M. Paul and B. Robinet, editors, *Proceedings of the International Symposium on Programming*, volume 167 of *LNCS*, pages 217–228, Toulouse, France, April 1984. Springer.

[332] Myers, Andrew C., Joseph A. Bank, and Barbara Liskov. Parameterized types for Java. In *ACM Symposium on Principles of Programming Languages (POPL), Paris, France*, pages 132–145, January 1997.

[333] Nadathur, Gopalan and Dale Miller. An overview of λProlog. In Robert A. Kowalski and Kenneth A. Bowen, editors, *Logic Programming: Proceedings of the Fifth International Conference and Symposium, Volume 1*, pages 810–827, MIT Press, Cambridge, Massachusetts, August 1988.

[334] Naur, Peter et al. Revised report on the algorithmic language Algol 60. *Communications of the ACM*, 6:1–17, January 1963.

[335] Necula, George C. Proof-carrying code. In *ACM Symposium on Principles of Programming Languages (POPL), Paris, France*, pages 106–119, 15–17 January 1997.

[336] Necula, George C. and Peter Lee. Safe kernel extensions without run-time checking. In *2nd Symposium on Operating Systems Design and Implementation (OSDI '96), October 28–31, 1996, Seattle, WA*, pages 229–243, Berkeley, CA, USA, October 1996. USENIX press.

[337] Necula, George C. and Peter Lee. Safe, untrusted agents using proof-carrying code. In G. Vigna, editor, *Mobile Agents and Security*, volume 1419 of *Lecture Notes in Computer Science*, pages 61–91. Springer-Verlag, 1998.

[338] Nelson, Greg, editor. *Systems Programming with Modula-3*. Prentice-Hall, 1991.

[339] Nipkow, Tobias and David von Oheimb. Java_{light} is type-safe — definitely. In *ACM Symposium on Principles of Programming Languages (POPL), San Diego, California*, pages 161–170, January 1998.

[340] O'Callahan, Robert and Daniel Jackson. Lackwit: A program understanding tool based on type inference. In *Proceedings of the 1997 International Conference on Software Engineering*, pages 338–348. ACM Press, 1997.

[341] Odersky, Martin. Functional nets. In *Proc. European Symposium on Programming (ESOP)*, pages 1–25. Springer-Verlag, 2000. Lecture Notes in Computer Science 1782.

[342] Odersky, Martin and Konstantin Läufer. Putting type annotations to work. In *ACM Symposium on Principles of Programming Languages (POPL), St. Petersburg Beach, Florida*, pages 54–67, St. Petersburg, Florida, January 21–24, 1996. ACM Press.

[343] Odersky, Martin, Martin Sulzmann, and Martin Wehr. Type inference with constrained types. *Theory and Practice of Object Systems*, 5(1):35–55, 1999. Summary in *Workshop on Foundations of Object-Oriented Languages (FOOL), informal proceedings*, 1997.

[344] Odersky, Martin and Philip Wadler. Pizza into Java: Translating theory into practice. In *ACM Symposium on Principles of Programming Languages (POPL), Paris, France*, pages 146–159, January 1997.

[345] Odersky, Martin and Christoph Zenger. Nested types. In *Workshop on Foundations of Object-Oriented Languages (FOOL 8)*, January 2001.

[346] Odersky, Martin, Christoph Zenger, and Matthias Zenger. Colored local type inference. *ACM SIGPLAN Notices*, 36(3):41–53, March 2001.

[347] O'Hearn, Peter W., Makoto Takeyama, A. John Power, and Robert D. Tennent. Syntactic control of interference revisited. In *MFPS XI, conference on Mathematical Foundations of Program Semantics*, volume 1 of *Electronic Notes in Theoretical Computer Science*. Elsevier, March 1995.

[348] O'Toole, James W. and David K. Gifford. Type reconstruction with first-class polymorphic values. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), Portland, Oregon*, pages 207–217, June 1989.

[349] Palsberg, Jens and Christina Pavlopoulou. From polyvariant flow information to intersection and union types. In *ACM Symposium on Principles of Programming Languages (POPL), San Diego, California*, pages 197–208, 1998.

[350] Palsberg, Jens and Michael I. Schwartzbach. *Object-Oriented Type Systems*. Wiley, 1994.

[351] Park, David. Concurrency and automata on infinite sequences. In P. Deussen, editor, *Proceedings of the 5th GI-Conference on Theoretical Computer Science*, volume 104 of *Lecture Notes in Computer Science*, pages 167–183. Springer-Verlag, Berlin, 1981.

[352] Paulin-Mohring, Christine. Extracting F_ω's programs from proofs in the calculus of constructions. In *ACM Symposium on Principles of Programming Languages (POPL), Austin, Texas*, pages 89–104, January 1989.

[353] Paulson, Laurence C. *ML for the Working Programmer*. Cambridge University Press, New York, NY, second edition, 1996.

[354] Peyton Jones, Simon L. and David R. Lester. *Implementing Functional Languages*. Prentice Hall, 1992.

[355] Pfenning, Frank. Partial polymorphic type inference and higher-order unification. In *ACM Symposium on Lisp and Functional Programming (LFP), Snowbird, Utah*, pages 153–163, July 1988. Also available as Ergo Report 88–048, School of Computer Science, Carnegie Mellon University, Pittsburgh.

[356] Pfenning, Frank. Elf: A language for logic definition and verified meta-programming. In *Fourth Annual Symposium on Logic in Computer Science*, pages 313–322, Pacific Grove, California, June 1989. IEEE Computer Society Press.

[357] Pfenning, Frank. On the undecidability of partial polymorphic type reconstruction. *Fundamenta Informaticae*, 19(1,2):185–199, 1993. Preliminary version available as Technical Report CMU-CS-92-105, School of Computer Science, Carnegie Mellon University, January 1992.

[358] Pfenning, Frank. Refinement types for logical frameworks. In Herman Geuvers, editor, *Informal Proceedings of the Workshop on Types for Proofs and Programs*, pages 285–299, Nijmegen, The Netherlands, May 1993.

[359] Pfenning, Frank. Elf: A meta-language for deductive systems. In A. Bundy, editor,

Proceedings of the 12th International Conference on Automated Deduction, pages 811–815, Nancy, France, June 1994. Springer-Verlag LNAI 814.

[360] Pfenning, Frank. The practice of logical frameworks. In Hélène Kirchner, editor, *Proceedings of the Colloquium on Trees in Algebra and Programming*, pages 119–134, Linköping, Sweden, April 1996. Springer-Verlag LNCS 1059. Invited talk.

[361] Pfenning, Frank. Logical frameworks. In Alan Robinson and Andrei Voronkov, editors, *Handbook of Automated Reasoning*. Elsevier, 1999.

[362] Pfenning, Frank. *Computation and Deduction*. Cambridge University Press, 2001.

[363] Pfenning, Frank and Peter Lee. Metacircularity in the polymorphic λ-calculus. *Theoretical Computer Science*, 89(1):137–159, 21 October 1991. Summary in *TAPSOFT '89, Proceedings of the International Joint Conference on Theory and Practice in Software Development, Barcelona, Spain*, pages 345–359, Springer-Verlag LNCS 352, March 1989.

[364] Pierce, Benjamin C. *Basic Category Theory for Computer Scientists*. MIT Press, 1991.

[365] Pierce, Benjamin C. *Programming with Intersection Types and Bounded Polymorphism*. PhD thesis, Carnegie Mellon University, December 1991. Available as School of Computer Science technical report CMU-CS-91-205.

[366] Pierce, Benjamin C. Bounded quantification is undecidable. *Information and Computation*, 112(1):131–165, July 1994. Also in Carl A. Gunter and John C. Mitchell, editors, *Theoretical Aspects of Object-Oriented Programming: Types, Semantics, and Language Design* (MIT Press, 1994). Summary in *ACM Symposium on Principles of Programming Languages (POPL), Albuquerque, New Mexico*.

[367] Pierce, Benjamin C. Even simpler type-theoretic foundations for OOP. Manuscript (circulated electronically), March 1996.

[368] Pierce, Benjamin C. Bounded quantification with bottom. Technical Report 492, Computer Science Department, Indiana University, 1997.

[369] Pierce, Benjamin C. Intersection types and bounded polymorphism. *Mathematical Structures in Computer Science*, 7(2):129–193, April 1997. Summary in *Typed Lambda Calculi and Applications*, March 1993, pp. 346–360.

[370] Pierce, Benjamin C. and Davide Sangiorgi. Typing and subtyping for mobile processes. In *Logic in Computer Science*, 1993. Full version in *Mathematical Structures in Computer Science* , Vol. 6, No. 5, 1996.

[371] Pierce, Benjamin C. and Martin Steffen. Higher-order subtyping. In *IFIP Working Conference on Programming Concepts, Methods and Calculi (PROCOMET)*, 1994. Full version in *Theoretical Computer Science*, vol. 176, no. 1–2, pp. 235–282, 1997 (corrigendum in TCS vol. 184 (1997), p. 247).

[372] Pierce, Benjamin C. and David N. Turner. Statically typed friendly functions via partially abstract types. Technical Report ECS-LFCS-93-256, University of Edinburgh, LFCS, April 1993. Also available as INRIA-Rocquencourt Rapport de Recherche No. 1899.

[373] Pierce, Benjamin C. and David N. Turner. Simple type-theoretic foundations for object-oriented programming. *Journal of Functional Programming*, 4(2):207–247, April 1994. Summary in *ACM Symposium on Principles of Programming Languages (POPL),*

Charleston, South Carolina, 1993.

[374] Pierce, Benjamin C. and David N. Turner. Local type argument synthesis with bounded quantification. Technical Report 495, Computer Science Department, Indiana University, January 1997.

[375] Pierce, Benjamin C. and David N. Turner. Local type inference. In *ACM Symposium on Principles of Programming Languages (POPL), San Diego, California*, 1998. Full version in *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 22(1), January 2000, pp. 1–44.

[376] Pierce, Benjamin C. and David N. Turner. Pict: A programming language based on the pi-calculus. In Gordon Plotkin, Colin Stirling, and Mads Tofte, editors, *Proof, Language and Interaction: Essays in Honour of Robin Milner*, pages 455–494. MIT Press, 2000.

[377] Pitts, Andrew M. Polymorphism is set theoretic, constructively. In Pitt, Poigné, and Rydeheard, editors, *Category Theory and Computer Science, Edinburgh*, pages 12–39. Springer-Verlag, 1987. LNCS volume 283.

[378] Pitts, Andrew M. Non-trivial power types can't be subtypes of polymorphic types. In *Fourth Annual Symposium on Logic in Computer Science, Pacific Grove, California*, pages 6–13. IEEE, June 1989.

[379] Pitts, Andrew M. Parametric polymorphism and operational equivalence. *Mathematical Structures in Computer Science*, 10:321–359, 2000.

[380] Plasmeijer, Marinus J. CLEAN: A programming environment based on term graph rewriting. *Theoretical Computer Science*, 194(1–2), March 1998.

[381] Plotkin, Gordon. Call-by-name, call-by-value, and the λ-calculus. *Theoretical Computer Science*, 1:125–159, 1975.

[382] Plotkin, Gordon and Martín Abadi. A logic for parametric polymorphism. In M. Bezem and J. F. Groote, editors, *Typed Lambda Calculi and Applications (TLCA), Utrecht, The Netherlands*, number 664 in Lecture Notes in Computer Science, pages 361–375. Springer-Verlag, March 1993.

[383] Plotkin, Gordon, Martín Abadi, and Luca Cardelli. Subtyping and parametricity. In *Proceedings of the Ninth IEEE Symposium on Logic in Computer Science*, pages 310–319, 1994.

[384] Plotkin, Gordon D. Lambda-definability and logical relations. Memorandum SAI–RM–4, University of Edinburgh, Edinburgh, Scotland, October 1973.

[385] Plotkin, Gordon D. LCF considered as a programming language. *Theoretical Computer Science*, 5:223–255, 1977.

[386] Plotkin, Gordon D. Lambda-definability in the full type hierarchy. In Jonathan P. Seldin and J. Roger Hindley, editors, *To H. B. Curry: Essays on Combinatory Logic, Lambda Calculus and Formalism*, pages 363–373. Academic Press, London, 1980.

[387] Plotkin, Gordon D. A structural approach to operational semantics. Technical Report DAIMI FN-19, Computer Science Department, Aarhus University, Aarhus, Denmark, 1981.

[388] Poll, Erik. Width-subtyping and polymorphic record update. Manuscript, June 1996.

[389] Pollack, Robert. Implicit syntax. Informal Proceedings of First Workshop on Logical Frameworks, Antibes, May 1990.

[390] Pollack, Robert. *The Theory of LEGO: A Proof Checker for the Extended Calculus of Constructions.* PhD thesis, University of Edinburgh, 1994.

[391] Pottier, François. Simplifying subtyping constraints. In *International Conference on Functional Programming (ICFP), Amsterdam, The Netherlands*, 1997.

[392] Pottinger, Garrell. A type assignment for the strongly normalizable λ-terms. In *To H. B. Curry: Essays on Combinatory Logic, Lambda Calculus, and Formalism*, pages 561–577. Academic Press, New York, 1980.

[393] Quine, Willard V. *Quiddities: An Intermittently Philosophical Dictionary.* Harvard University Press, Cambridge, MA, 1987.

[394] Ramsey, Frank P. The foundations of mathematics. *Proceedings of the London Mathematical Society, Series 2*, 25(5):338–384, 1925. Reprinted in Braithwaite [55].

[395] Ranta, Aarne. *Type-Theoretical Grammar.* Clarendon Press, Oxford, 1995.

[396] Reade, Chris. *Elements of Functional Programming.* International Computer Science Series. Addison-Wesley, Wokingham, England, 1989.

[397] Reddy, Uday S. Objects as closures: Abstract semantics of object oriented languages. In *ACM Symposium on Lisp and Functional Programming (LFP), Snowbird, Utah*, pages 289–297, Snowbird, Utah, July 1988.

[398] Relax. Document Description and Processing Languages — Regular Language Description for XML (RELAX) — Part 1: RELAX Core. Technical Report DTR 22250-1, ISO/IEC, October 2000.

[399] Rémy, Didier. Typechecking records and variants in a natural extension of ML. In *ACM Symposium on Principles of Programming Languages (POPL), Austin, Texas*, pages 242–249, January 1989. Long version in Carl A. Gunter and John C. Mitchell, editors, *Theoretical Aspects of Object-Oriented Programming: Types, Semantics, and Language Design* (MIT Press, 1994).

[400] Rémy, Didier. *Algèbres Touffues. Application au Typage Polymorphe des Objets Enregistrements dans les Langages Fonctionnels.* PhD thesis, Université Paris VII, 1990.

[401] Rémy, Didier. Extending ML type system with a sorted equational theory. Research Report 1766, Institut National de Recherche en Informatique et Automatisme, Rocquencourt, BP 105, 78 153 Le Chesnay Cedex, France, 1992.

[402] Rémy, Didier. Projective ML. In *ACM Symposium on Lisp and Functional Programming (LFP)*, pages 66–75, 1992.

[403] Rémy, Didier. Typing record concatenation for free. In *ACM Symposium on Principles of Programming Languages (POPL), Albuquerque, New Mexico*, January 1992. Also in Carl A. Gunter and John C. Mitchell, editors, *Theoretical Aspects of Object-Oriented Programming: Types, Semantics, and Language Design* (MIT Press, 1994).

[404] Rémy, Didier. Programming objects with ML-ART: An extension to ML with abstract and record types. In Masami Hagiya and John C. Mitchell, editors, *International Symposium on Theoretical Aspects of Computer Software (TACS)*, pages 321–346, Sendai, Japan, April 1994. Springer-Verlag.

[405] Rémy, Didier. *Des enregistrements aux objets.* Mémoire d'habilitation à diriger des recherches, Université de Paris 7, 1998. In English, except for introductory chapter; includes Rémy [399] and Rémy [402].

[406] Rémy, Didier and Jérôme Vouillon. Objective ML: An effective object-oriented extension to ML. *Theory and Practice of Object Systems*, 4(1):27–50, 1998. Summary in *ACM Symposium on Principles of Programming Languages (POPL), Paris, France*, 1997.

[407] Reynolds, John. Three approaches to type structure. In *Mathematical Foundations of Software Development*. Springer-Verlag, 1985. Lecture Notes in Computer Science No. 185.

[408] Reynolds, John C. Towards a theory of type structure. In *Proc. Colloque sur la Programmation*, pages 408–425, New York, 1974. Springer-Verlag LNCS 19.

[409] Reynolds, John C. User-defined types and procedural data structures as complementary approaches to data abstraction. In Stephen A. Schuman, editor, *New Directions in Algorithmic Languages 1975*, pages 157–168, Rocquencourt, France, 1975. IFIP Working Group 2.1 on Algol, INRIA. Reprinted in Gries [187], pages 309–317 and Gunter and Mitchell [189], pages 13–23.

[410] Reynolds, John C. Syntactic control of interference. In *ACM Symposium on Principles of Programming Languages (POPL), Tucson, Arizona*, pages 39–46, 1978. Reprinted in O'Hearn and Tennent, *ALGOL-like Languages*, vol. 1, pages 273–286, Birkhäuser, 1997.

[411] Reynolds, John C. Using category theory to design implicit conversions and generic operators. In N. D. Jones, editor, *Proceedings of the Aarhus Workshop on Semantics-Directed Compiler Generation*, number 94 in Lecture Notes in Computer Science. Springer-Verlag, January 1980. Also in Carl A. Gunter and John C. Mitchell, editors, *Theoretical Aspects of Object-Oriented Programming: Types, Semantics, and Language Design* (MIT Press, 1994).

[412] Reynolds, John C. *The Craft of Programming*. Prentice-Hall International, London, 1981.

[413] Reynolds, John C. Types, abstraction, and parametric polymorphism. In R. E. A. Mason, editor, *Information Processing 83*, pages 513–523, Amsterdam, 1983. Elsevier Science Publishers B. V. (North-Holland).

[414] Reynolds, John C. Polymorphism is not set-theoretic. In G. Kahn, D. B. MacQueen, and G. D. Plotkin, editors, *Semantics of Data Types*, volume 173 of *Lecture Notes in Computer Science*, pages 145–156, Berlin, 1984. Springer-Verlag.

[415] Reynolds, John C. Preliminary design of the programming language Forsythe. Technical Report CMU-CS-88-159, Carnegie Mellon University, June 1988. Reprinted in O'Hearn and Tennent, *ALGOL-like Languages*, vol. 1, pages 173–233, Birkhäuser, 1997.

[416] Reynolds, John C. Syntactic control of interference, part 2. Report CMU-CS-89-130, Carnegie Mellon University, April 1989.

[417] Reynolds, John C. Introduction to part II, polymorphic lambda calculus. In Gérard Huet, editor, *Logical Foundations of Functional Programming*, pages 77–86. Addison-Wesley, Reading, Massachusetts, 1990.

[418] Reynolds, John C. The coherence of languages with intersection types. In T. Ito and A. R. Meyer, editors, *Theoretical Aspects of Computer Software (Sendai, Japan)*, number 526 in Lecture Notes in Computer Science, pages 675–700. Springer-Verlag, September 1991.

[419] Reynolds, John C. Normalization and functor categories. In Olivier Danvy and Peter Dybjer, editors, *Preliminary Proceedings of the 1998 APPSEM Workshop on Normalization by Evaluation, NBE '98, (Chalmers, Sweden, May 8-9, 1998)*, number NS-98-1 in BRICS Note Series, Department of Computer Science, University of Aarhus, May 1998.

[420] Reynolds, John C. *Theories of Programming Languages.* Cambridge University Press, 1998.

[421] Reynolds, John C. and Gordon Plotkin. On functors expressible in the polymorphic typed lambda calculus. *Information and Computation*, 105(1):1–29, 1993. Summary in Huet [229].

[422] Robinson, Edmund and Robert Tennent. Bounded quantification and record-update problems. Message to Types electronic mail list, October 1988.

[423] Robinson, J. Alan. Computational logic: The unification computation. *Machine Intelligence*, 6:63–72, 1971.

[424] Russell, Bertrand. Letter to Frege, 1902. Reprinted (in English) in J. van Heijenort, editor, *From Frege to Gödel: A Source Book in Mathematical Logic, 1879–1931*; Harvard University Press, Cambridge, MA, 1967; pages 124–125.

[425] Schaffert, Justin Craig. A formal definition of CLU. Master's thesis, MIT, January 1978. MIT/LCS/TR-193.

[426] Scheifler, Robert William. A denotational semantics of CLU. Master's thesis, MIT, May 1978. MIT/LCS/TR-201.

[427] Schmidt, David A. *Denotational Semantics: A Methodology for Language Development.* Allyn and Bacon, 1986.

[428] Schmidt, David A. *The Structure of Typed Programming Languages.* MIT Press, 1994.

[429] Schönfinkel, Moses. Über die Bausteine der mathematischen Logik. *Mathematische Annalen*, 92:305–316, 1924. Translated into English and republished as "On the building blocks of mathematical logic" in van Heijenoort [475], pp. 355–366.

[430] Scott, Michael L. *Programming Language Pragmatics.* Morgan Kaufmann, 1999.

[431] Severi, Paula and Erik Poll. Pure type systems with definitions. In *Proceedings of Logical Foundations of Computer Science (LFCS)*, pages 316–328. Springer-Verlag, 1994. LNCS volume 813.

[432] Shalit, Andrew. *The Dylan Reference Manual: The Definitive Guide to the New Object-Oriented Dynamic Language.* Addison-Wesley, Reading, Mass., 1997.

[433] Shields, Mark. *Static Types for Dynamic Documents.* PhD thesis, Department of Computer Science, Oregon Graduate Institute, February 2001.

[434] Simmons, Harold. *Derivation and Computation : Taking the Curry-Howard Correspondence Seriously.* Number 51 in Cambridge Tracts in Theoretical Computer Science. Cambridge University Press, 2000.

[435] Smith, Frederick, David Walker, and Greg Morrisett. Alias types. In Gert Smolka, editor, *Ninth European Symposium on Programming*, volume 1782 of *Lecture Notes in Computer Science*, pages 366–381. Springer-Verlag, April 2000.

[436] Smith, Jan, Bengt Nordström, and Kent Petersson. *Programming in Martin-Löf's Type Theory. An Introduction.* Oxford University Press, 1990.

[437] Solomon, Marvin. Type definitions with parameters. In *ACM Symposium on Principles of Programming Languages (POPL), Tucson, Arizona*, pages 31–38, January 23–25, 1978.

[438] Sommaruga, Giovanni. *History and Philosophy of Constructive Type Theory*, volume 290 of *Synthese Library*. Kluwer Academic Pub., 2000.

[439] Somogyi, Zoltan, Fergus Henderson, and Thomas Conway. The execution algorithm of Mercury, an efficient purely declarative logic programming language. *Journal of Logic Programming*, 29(1–3):17–64, October–November 1996.

[440] Sørensen, Morten Heine and Paweł Urzyczyn. Lectures on the Curry-Howard isomorphism. Technical Report 98/14 (= TOPPS note D-368), DIKU, Copenhagen, 1998.

[441] Statman, Richard. Completeness, invariance and λ-definability. *Journal of Symbolic Logic*, 47(1):17–26, 1982.

[442] Statman, Richard. Equality between functionals, revisited. In *Harvey Friedman's Research on the Foundations of Mathematics*, pages 331–338. North-Holland, Amsterdam, 1985.

[443] Statman, Richard. Logical relations and the typed λ-calculus. *Information and Control*, 65(2–3):85–97, May–June 1985.

[444] Steffen, Martin. *Polarized Higher-Order Subtyping*. PhD thesis, Universität Erlangen-Nürnberg, 1998.

[445] Stone, Christopher A. and Robert Harper. Deciding type equivalence in a language with singleton kinds. In *ACM Symposium on Principles of Programming Languages (POPL), Boston, Massachusetts*, pages 214–227, January 19–21, 2000.

[446] Strachey, Christopher. Fundamental concepts in programming languages. Lecture Notes, International Summer School in Computer Programming, Copenhagen, August 1967. Reprinted in *Higher-Order and Symbolic Computation*, 13(1/2), pp. 1–49, 2000.

[447] Stroustrup, Bjarne. *The C++ Programming Language*. Addison Wesley Longman, Reading, MA, third edition, 1997.

[448] Studer, Thomas. Constructive foundations for Featherweight Java. In R. Kahle, P. Schroeder-Heister, and R. Stärk, editors, *Proof Theory in Computer Science*. Springer-Verlag, 2001. Lecture Notes in Computer Science, volume 2183.

[449] Sumii, Eijiro and Benjamin C. Pierce. Logical relations for encryption. In *Computer Security Foundations Workshop*, June 2001. Submitted (by invitation) to *Journal of Computer Security*.

[450] Sussman, Gerald Jay and Guy Lewis Steele, Jr. Scheme: an interpreter for extended lambda calculus. MIT AI Memo 349, Massachusetts Institute of Technology, December 1975. Reprinted, with a foreword, in *Higher-Order and Symbolic Computation*, 11(4), pp. 405–439, 1998.

[451] Syme, Don. Proving Java type soundness. Technical Report 427, Computer Laboratory, University of Cambridge, June 1997.

[452] Tait, William W. Intensional interpretations of functionals of finite type I. *Journal of Symbolic Logic*, 32(2):198–212, June 1967.

[453] Tait, William W. A realizability interpretation of the theory of species. In R. Parikh, editor, *Logic Colloquium*, volume 453 of *Lecture Notes in Mathematics*, pages 240–251,

Boston, 1975. Springer-Verlag.

[454] Talpin, Jean-Pierre and Pierre Jouvelot. The type and effects discipline. In *Proc. IEEE Symp. on Logic in Computer Science*, pages 162–173, 1992.

[455] Tarditi, David, Greg Morrisett, Perry Cheng, Christopher Stone, Robert Harper, and Peter Lee. TIL: A type-directed optimizing compiler for ML. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), Philadephia, Pennsylvania*, pages 181–192, May 21–24 1996.

[456] Tarski, Alfred. A lattice-theoretical fixpoint theorem and its applications. *Pacific Journal of Mathematics*, 5:285–309, 1955.

[457] Tennent, Robert D. *Principles of Programming Languages.* Prentice-Hall, 1981.

[458] Terlouw, J. Een nadere bewijstheoretische analyse van GSTTs. Manuscript, University of Nijmegen, Netherlands, 1989.

[459] Thatte, Satish R. Quasi-static typing (preliminary report). In *ACM Symposium on Principles of Programming Languages (POPL), San Francisco, California*, pages 367–381, 1990.

[460] Thompson, Simon. *Type Theory and Functional Programming.* Addison Wesley, 1991.

[461] Thompson, Simon. *Haskell: The Craft of Functional Programming.* Addison Wesley, 1999.

[462] Tiuryn, Jerzy. Type inference problems: A survey. In B. Rovan, editor, *Mathematical Foundations of Computer Science 1990, Banskà Bystrica, Czechoslovakia*, volume 452 of *Lecture Notes in Computer Science*, pages 105–120. Springer-Verlag, New York, NY, 1990.

[463] Tofte and Birkedal. A region inference algorithm. *ACMTOPLAS: ACM Transactions on Programming Languages and Systems*, 20, 1998.

[464] Tofte, Mads. Type inference for polymorphic references. *Information and Computation*, 89(1), November 1990.

[465] Tofte, Mads and Jean-Pierre Talpin. Implementing the call-by-value lambda-calculus using a stack of regions. In *ACM Symposium on Principles of Programming Languages (POPL), Portland, Oregon*, January 1994.

[466] Tofte, Mads and Jean-Pierre Talpin. Region-based memory management. *Information and Computation*, 132(2):109–176, 1 February 1997.

[467] Trifonov, Valery and Scott Smith. Subtyping constrained types. In *Proceedings of the Third International Static Analysis Symposium*, volume 1145 of *LNCS*, pages 349–365. Springer-Verlag, September 1996.

[468] Turner, David N., Philip Wadler, and Christian Mossin. Once upon a type. In *ACM Symposium on Functional Programming Languages and Computer Architecture (FPCA)*, San Diego, California, 1995.

[469] Turner, Raymond. *Constructive Foundations for Functional Languages.* McGraw Hill, 1991.

[470] Ullman, Jeffrey D. *Elements of ML Programming.* Prentice-Hall, ML97 edition, 1997.

[471] Ungar, David and Randall B. Smith. Self: The power of simplicity. In *ACM Symposium on Object Oriented Programming: Systems, Languages, and Applications (OOPSLA)*, pages 227–241, 1987.

[472] U.S. Dept. of Defense. *Reference Manual for the Ada Programming Language.* GPO 008-000-00354-8, 1980.

[473] van Benthem, Johan. *Language in Action: Categories, Lambdas, and Dynamic Logic.* MIT Press, 1995.

[474] van Benthem, Johan F. A. K. and Alice Ter Meulen, editors. *Handbook of Logic and Language.* MIT Press, 1997.

[475] van Heijenoort, Jan, editor. *From Frege to Gödel.* Harvard University Press, Cambridge, Massachusetts, 1967.

[476] van Wijngaarden, Adriaan, B. J. Mailloux, J. E. L. Peck, C. H. A. Koster, M. Sintzoff, C. H. Lindsey, L. G. L. T. Meertens, and R. G. Fisker. Revised report on the algorithmic language ALGOL 68. *Acta Informatica*, 5(1-3):1-236, 1975.

[477] Vouillon, Jérôme. *Conception et réalisation d'une extension du langage ML avec des objets.* PhD thesis, Université Paris 7, October 2000.

[478] Vouillon, Jérôme. Combining subsumption and binary methods: An object calculus with views. In *ACM Symposium on Principles of Programming Languages (POPL), London, England*, 2001.

[479] Wadler, Philip. Theorems for free! In *Functional Programming Languages and Computer Architecture*, pages 347-359. ACM Press, September 1989. Imperial College, London.

[480] Wadler, Philip. Linear types can change the world. In *TC 2 Working Conference on Programming Concepts and Methods (Preprint)*, pages 546-566, 1990.

[481] Wadler, Philip. Is there a use for linear logic? In *Proceedings of ACM Symposium on Partial Evaluation and Semantics-Based Program Manipulation*, pages 255-273, 1991.

[482] Wadler, Philip. New languages, old logic. *Dr. Dobbs Journal*, December 2000.

[483] Wadler, Philip. The Girard-Reynolds isomorphism. In Naoki Kobayashi and Benjamin Pierce, editors, *Theoretical Aspects of Computer Software (TACS), Sendai, Japan*, Lecture Notes in Computer Science. Springer-Verlag, 2001.

[484] Wadler, Philip and Stephen Blott. How to make *ad-hoc* polymorphism less *ad hoc*. In *ACM Symposium on Principles of Programming Languages (POPL), Austin, Texas*, pages 60-76, 1989.

[485] Wadsworth, Christopher P. *Semantics and pragmatics of the lambda-calculus.* PhD thesis, Programming Research Group, Oxford University, 1971.

[486] Wand, Mitchell. Finding the source of type errors. *13th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages (POPL)*, pages 38-43, 1986.

[487] Wand, Mitchell. Complete type inference for simple objects. In *Proceedings of the IEEE Symposium on Logic in Computer Science*, Ithaca, NY, June 1987.

[488] Wand, Mitchell. Corrigendum: Complete type inference for simple objects. In *Proceedings of the IEEE Symposium on Logic in Computer Science*, 1988.

[489] Wand, Mitchell. Type inference for objects with instance variables and inheritance. Technical Report NU-CCS-89-2, College of Computer Science, Northeastern University, February 1989. Also in Carl A. Gunter and John C. Mitchell, editors, *Theoretical Aspects of Object-Oriented Programming: Types, Semantics, and Language Design* (MIT Press, 1994).

[490] Wand, Mitchell. Type inference for record concatenation and multiple inheritance. In *Fourth Annual IEEE Symposium on Logic in Computer Science*, pages 92–97, Pacific Grove, CA, June 1989.

[491] Weis, Pierre, María-Virginia Aponte, Alain Laville, Michel Mauny, and Ascánder Suárez. The CAML reference manual, Version 2.6. Technical report, Projet Formel, INRIA-ENS, 1989.

[492] Wells, Joe B. Typability and type checking in the second-order λ-calculus are equivalent and undecidable. In *Proceedings of the Ninth Annual IEEE Symposium on Logic in Computer Science (LICS)*, pages 176–185, 1994.

[493] Whitehead, Alfred North and Bertrand Russell. *Principia Mathematica*. Cambridge University Press, Cambridge, 1910. Three volumes (1910; 1912; 1913).

[494] Wickline, Philip, Peter Lee, Frank Pfenning, and Rowan Davies. Modal types as staging specifications for run-time code generation. *ACM Computing Surveys*, 30(3es), September 1998. Article 8.

[495] Wille, Christoph. *Presenting C#*. SAMS Publishing, 2000.

[496] Winskel, Glynn. *The Formal Semantics of Programming Languages: An Introduction*. MIT Press, 1993.

[497] Wirth, Niklaus. The programming language Pascal. *Acta Informatica*, 1(1):35–63, 1971.

[498] Wright, Andrew K. Typing references by effect inference. In Bernd Krieg-Bruckner, editor, *ESOP '92, 4th European Symposium on Programming, Rennes, France*, volume 582 of *Lecture Notes in Computer Science*, pages 473–491. Springer-Verlag, New York, N.Y., 1992.

[499] Wright, Andrew K. Simple imperative polymorphism. *Lisp and Symbolic Computation*, 8(4):343–355, 1995.

[500] Wright, Andrew K. and Matthias Felleisen. A syntactic approach to type soundness. *Information and Computation*, 115(1):38–94, 15 November 1994.

[501] Xi, Hongwei and Robert Harper. A dependently typed assembly language. In *International Conference on Functional Programming (ICFP), Firenze, Italy*, 2001.

[502] Xi, Hongwei and Frank Pfenning. Eliminating array bound checking through dependent types. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), Montreal, Canada*, pages 249–257, 1998.

[503] Xi, Hongwei and Frank Pfenning. Dependent types in practical programming. In *ACM Symposium on Principles of Programming Languages (POPL), San Antonio, Texas*, ACM SIGPLAN Notices, pages 214–227, 1999.

[504] XML 1998. Extensible markup language (XML™), February 1998. XML 1.0, W3C Recommendation, `http://www.w3.org/XML/`.

[505] XS 2000. XML Schema Part 0: Primer, W3C Working Draft. `http://www.w3.org/TR/xmlschema-0/`, 2000.

[506] Yelick, Kathy, Luigi Semenzato, Geoff Pike, Carleton Miyamoto, Ben Liblit, Arvind Krishnamurthy, Paul Hilfinger, Susan Graham, David Gay, Phil Colella, and Alex Aiken. Titanium: A high-performance Java dialect. *Concurrency: Practice and Experience*, 10 (11-13):825–836, September 1998. Special Issue: Java for High-performance Network Computing.

[507] Zwanenburg, Jan. Pure type systems with subtyping. In J.-Y. Girard, editor, *Typed Lambda Calculus and Applications (TLCA)*, pages 381–396. Springer-Verlag, 1999. Lecture Notes in Computer Science, volume 1581.

創造性の秘訣は、情報源の隠し方を知ることである。 —*Albert Einstein*

訳語集

ギリシア文字

μ 型	μ-type
μ 畳み込み	μ-folding
π 計算	pi-calculus

C

Church 数	Church numeral
Church スタイル	Church-style
Church 表現	Church encoding
Curry-Howard 対応	Curry-Howard correspondence
Curry スタイル	Curry-style

F

F 有界量化	F-bounded quantification

L

let 多相	let-polymorphism

M

ML 式多相	ML-style polymorphism

Y

Y コンビネータ	Y-combinator

ア

値	value
値制限	value restriction
値呼び	call by value
アップキャスト	upcast
アドホック多相	ad-hoc polymorphism
アルゴリズム的型付け	algorithmic typing
アルゴリズム的型付け関係	algorithmic typing relation
アルゴリズム的部分型関係	algorithmic subtyping relation
アルゴリズム的部分型付け	algorithmic subtyping
アルファ変換	alpha-conversion
安全性	safety
安全代入の原則	principle of safe substitution
アンボックス化された	unboxed

イ

行き詰まり（状態）	stuck (state)
委譲	delegation
依存カインド	dependent kind
依存型	dependent type
依存関数型	dependent function type
位置（ストアでの）	location
一元カインド	singleton kind
一様	uniform
一級	first-class
一級市民	first-class citizen
一級多相	first-class polymorphism
一般化（型変数の）	generalization
意味	meaning
意味領域	semantic domain
意味論	semantics
色付き局所型推論	colored local type inference
インスタンス	instance
インスタンス化する	instantiate
インスタンス変数	instance variable
インターフェイス	interface
隠蔽表現型	hidden representation type

ウ

写す	map

エ

エラー	error

オ

オーバーライド	override
オーバーロード	overloading
オープンな再帰	open recursion
おかしくなる（項が/プログラムが）	go wrong
オブジェクト	object
オブジェクト計算	object calculus
オブジェクトコンストラクタ	object constructor
オブジェクト指向	object-oriented
オブジェクト同一性	object identity
オプション値	optional value
重み	weight
愚かなキャスト	stupid cast

カ

カーネル部分型付け規則	kernel subtyping rule
解釈関数	interpretation function
階層化されている	stratified
外部言語	external language
解放（記憶域の）	deallocation
カインド	kind
カインド付け	kinding
書き換え可能な	mutable
可逆	invertible
拡張可能バリアント型	extensible variant type
拡張する（クラスを）	extend
拡張静的検査	Extended Static Checking
確保（参照の/記憶域の）	allocation
各要素・各引数での	pointwise
下限	greatest lower bound
可算	countable
可述的	predicative
型	type
型演算子	type operator
型環境	type environment
型強制	coercion
型クラス	type class
型検査	typechecking
型構築子	type constructor
型再構築	type reconstruction
型システム	type system
型指定	type ascription
型主導部分評価	type-directed partial evaluation
型消去	type erasure
型推論	type inference
型スキーム	type scheme
型代入	type substitution
型タグ	type tag
型注釈	type annotation
型抽象	type abstraction
型付け	typing
型付け可能	typable
型付け関係	typing relation
型付け規則	typing rule
型付け導出	typing derivation
型付け判断式	typing statement

型付け文脈	typing context
型適用	type application
型等価性	type equivalence
型無し	untyped
型分解子	type destructor
型変数	type variable
型名	type name
型理論	type theory
型露出	type exposure
型渡し意味論	type-passing semantics
型割り当て体系	type-assignment system
カット除去	cut elimination
合併型	union type
仮定	assumption
カプセル化	encapsulation
ガベージコレクション	garbage collection
カリー化	currying
仮引数	formal parameter
環境	environment
関係	relation
関数	function
完全	complete
完全帰納法	complete induction
完全抽象	full abstraction
完全ベータ簡約	full beta-reduction
完全有界量化	completely bounded quantification
冠頭多相	prenex polymorphism
簡約	reduction
簡約可能候補	reducible candidate
簡約基	redex

キ

木型	tree type
規則のスキーマ	rule schema
基底（生成関数 F の）	F-ground
起動記録	activation record
帰納法	induction
記法/表記	notation
基本型	base type
逆転補題	inversion lemma
キャスト	cast(ing)
狭化	narrowing
強化	strengthening
境界検査（配列の）	array-bounds checking
強正規化	strong normalization
強二項演算	strong binary operation
共変	covariant
共有状態	shared state
局所型再構築	local type reconstruction
局所型推論	local type inference
極性	polarity

ク

空集合	empty set
具象構文	concrete syntax
具体化する（型を）	instantiate
組	tuple
クラス	class

ケ

計算規則	computation rule
計算的作用	computational effect
形式手法	formal method
継承	inheritance
継続	continuation
軽量形式手法	lightweight formal method
結合規則	associativity
決定可能	decidable
決定手続き	decision procedure
原子型	atomic type
健全	sound
限定型	qualified type
限定的でない（代入が）	less specific
圏論	category theory

コ

項	term
高階型演算子	higher-order type operator
高階関数	higher-order function
高階有界量化子	higher-order bounded quantifiers
高階量化	higher-order quantification
降下列	decreasing chain
交差型	intersection type
後者値	successor
後者関数	successor
合成性	compositionality
構成的型理論	constructive type theory
構成的論理	constructive logic
構成により正しい	correct by construction
構造的型システム	structural type system
構造的帰納法	structural induction
構造的操作的意味論	structural operational semantics
構造的展開	structural unfolding
構築子	constructor
恒等関数	identity function
合同規則	congruence rule
構文解析器	parser
構文主導	syntax directed
構文による干渉の制御	syntactic control of interference
項変数	term variable
公理	axiom
公理的意味論	axiomatic semantics
合流性	confluent
ゴール	goal
混置	mixfix
コンビネータ	combinator
コンビネータ論理	combinatory logic

サ

再帰型	recursive type
最左	leftmost
最小型	minimal type
サイズ	size
最汎単一化子	most general unifier
サブクラス	subclass
サンク	thunk
算術	arithmetic
算術式	arithmetic expression
参照	reference
参照先の値を取り出す	dereference

シ

シェアリング	sharing
シェイプ型	shape type
ジェネリクス	generics
式	expression
字句解析器	lexical analyzer
次元解析	dimension analysis
支持関数	support function
支持グラフ	support graph
支持されている（生成関数 F に）	F-supported
辞書式帰納法	lexicographic induction
辞書式順序	lexicographic order
自然意味論	natural semantics
自然演繹	natural deduction
自然数	natural number
下に有界	bounded below
実行時エラー	run-time error
実行時型エラー	run-time type error

実行時コード生成	run-time code generation
実行時モニタリング	run-time monitoring
実引数	actual parameter
シフト	shifting
弱型変数	weak type variable
弱頭部簡約	weak head reduction
弱二項演算	weak binary operation
写像	mapping
弱化	weakening
自由（変数の出現が）	free
充足する	satisfy
柔軟な組	flexible tuple
自由変数	free variable
縮小的	contractive
出現検査	occur check
述語	predicate
主部簡約	subject reduction
主部展開	subject expansion
主部評価	subject evaluation
主要解	principal solution
主要型	principal type
主要型付け	principal typing
主要単一化子	principal unifier
順序集合	ordered set
純粋型システム	pure type system (PTS)
純粋関数的	purely functional
純粋でない	impure
純粋な	pure
純粋なラムダ計算	pure lambda-calculus
上位型	supertype
昇格する（型を）	promote
上限	least upper bound
条件式	conditional (expression)
詳細化関数	elaboration function
詳細型	refinement type
小ステップ操作的意味論	small-step operational semantics
小ステップ評価	small-step evaluation
状態	state
状態型（オブジェクトの）	state type
証人型	witness type
証明論	proof theory
省略/略記	abbreviation
除去規則	elimination rule
所属性検査	membership checking
処理する（例外を）	handle
進行	progress
真の型	proper type

ス

推移的	transitive
推移的閉包	transitive closure
推論規則	inference rule
数値	numeric value
スーパークラス	superclass
スコープ	scope
スタック	stack
ストア	store
ストア型付け	store typing
ストアでの位置	store location

セ

正格	strict
正規化	normalization
正規化可能	normalizable
正規形	normal form
正規順序	normal order
整形表示	pretty printing
整合性	coherence
整合的（生成関数 F について）	F-consistent
生成関数	generating function
生成集合	generating set
生成補題	generation lemma
整礎（順序が/集合が）	well founded
正則型	regular type
静的距離	static distance
正の	positive
正の部分型付け	positive subtyping
制約型	constraint type
制約型付け	constraint typing
制約型付け関係	constraint typing relation
制約集合	constraint set
遷移関数	transition function
全域関数	total function
線形論理	linear logic
宣言的	declarative
先行的	eager
潜在的型付け	latent typing
前者値	predecessor
前者関数	predecessor
前順序	preorder
全順序	total order
全称型	universal type
全称量化子	universal quantifier

ソ

操作的意味論	operational semantics
送出する（例外を）	raise
総称性	genericity
双模倣性	bisimulation
束縛	bound
束縛子	binder
存在型	existential type
存在型被単一化子	existential unificand
存在量化子	existential quantifier

タ

対象言語	object language
対称的	symmetric
大ステップ操作的意味論	big-step operational semantics
大ステップ評価	big-step evaluation
代入	substitution
代入補題	substitution lemma
ダイヤモンド性	diamond property
ダウンキャスト	downcast
多型	polytype
タグ付け	tagging
タグなしガベージコレクション	tag-free garbage collection
多重帰納法	nested induction
多重継承	multiple inheritance
多重メソッドディスパッチ	multi-method dispatch
多ステップ評価	multi-step evaluation
多相恒等関数	polymorphic identity function
多相再帰	polymorphic recursion
多相再パッケージ化	polymorphic repacking
多相性	polymorphism
多相的な	polymorphic
多相ラムダ計算	polymorphic lambda-calculus
多相（レコード）更新	polymorphic (record) update
正しい形式を持つ	well formed
正しく型付けされていない	ill-typed
正しく型付けされている	well typed
脱糖衣	desugaring
単一化する	unify
単型	monotype
単純階型理論	simple theory of types
単純型	simple type
単純型付きラムダ計算	simply typed lambda-calculus
単調関数	monotone function

チ

値域	range
遅延束縛	late binding
遅延評価	lazy
逐次実行	sequencing
チャネル型	channel type
中間言語	intermediate language
抽象/抽象化	abstraction
抽象機械	abstract machine
抽象境界	abstraction boundary
抽象構文	abstract syntax
抽象構文木	abstract syntax tree
抽象データ型	abstract data type (ADT)
直積	product
直列的に受け渡し	threading
直和	disjoint union

テ

定義域	domain
定義による等価性	definitional equivalence
停止尺度	termination measure
デカルト積	cartesian product
手続きによる抽象化	procedural abstraction

ト

糖衣構文	syntactic sugar
等価性検査	equivalence check
同型	isomorphic
同型再帰	iso-recursive
等式単一化	equational unification
導出可能	derivable
導出木	derivation tree
導出に関する帰納法	induction on derivation
到達可能	reachable
同値	equivalent
同値関係	equivalence
同値再帰	equi-recursive
動的型付け	dynamic typing
動的型テスト	dynamic type-testing
動的ディスパッチ	dynamic dispatch
導入規則	introduction rule
トークン	token
閉じている（項が）	closed
閉じている（生成関数 F について）	F-closed
貪欲型推論	greedy type inference

ナ

内部クラス	inner class
内部言語	internal language
内包的多相	intensional polymorphism
内包表記	comprehension notation
名無し項	nameless term
名無し表現	nameless representation
名前付け文脈	naming context
名前的型システム	nominal type system
名前呼び	call by name
生の μ 型	raw μ-types
並べ替え	permutation
何でもよい	don't care

ニ

二階ラムダ計算	second-order lambda-calculus
二項演算	binary operation
二項関係	binary relation
入射	inject

ハ

排中律	excluded middle
破壊的代入	assignment
派生形式	derived form
パターン	pattern
パターン型付け	pattern typing
パターンマッチ	pattern matching
パッケージ	package
発散コンビネータ	divergent combinator
発散する	diverge
ハッシュコンシング	hash consing
幅部分型付け	width subtyping
パラメータ	parameter
パラメータ性	parametricity
パラメータ多相	parametric polymorphism
バリアント	variant
バリアント型	variant type
反射的	reflexive
反射的推移的閉包	reflexive and transitive closure
反射的閉包	reflexive closure
半順序	partial order
反対称的	antisymmetric
半単一化	semi-unification
半抽象型	partially abstract type
範疇文法	categorial grammar
判定	test
反変	contravariant

ヒ

ヒープ	heap
非解釈の基本型	uninterpreted base type
非可述的	impredicative
非可述的多相	impredicative polymorphism
引数	parameter/argument
非正格な	non-strict
必要呼び	call by need
評価	evaluation
評価関係	evaluation relation
評価関係式	evaluation statement
評価戦略	evaluation strategy
評価による正規化	normalization by evaluation
評価判断式	evaluation judgment
評価文脈	evaluation context
表記/記法	notation
表現	encoding
表現	representation
表現型	representation type
表現非依存性	representation independence
表現力	expressive power/expressiveness
表示的意味論	denotational semantics
標準形	canonical form
表層構文	surface syntax

フ

ファイナライザ	finalizer
フィールド参照	field access
ブール値	boolean
深さ（項の）	depth
深さ部分型付け	depth subtyping
複合的な型	compound type
副作用	side effect
付帯条件	side condition
二つ組	pair
不動点	fixed point
不動点コンビネータ	fixed-point combinator
負の	negative
部分型	subtype
部分型多相	subtype polymorphism
部分型付け	subtyping
部分型付け制約	subtyping constraint
部分型付け導出	subtyping derivation
部分型付け判断式	subtyping statement

部分関数	partial function
部分木	subtree
部分集合意味論	subset semantics
部分的型再構築	partial type reconstruction
部分同値関係	partial equivalence relation
部分評価	partial evaluation
普遍集合	universal set
不変条件	invariant
普遍領域	universal domain
ぶら下がり参照	dangling reference
フル部分型付け規則	full subtyping rule
振る舞い等価	behaviorally equivalent
プロセス	process
分解子	destructor
分岐階型理論	ramified theory of types
文法	grammar
文脈	context

ヘ

並行簡約	parallel reduction
平坦化されたデータ構造	flattened data structures
閉包性	closure property
ベータ簡約	beta-reduction
冪カインド	power kind
冪型	power type
冪集合	powerset
別名解析	alias analysis
別名付け	aliasing
変数	variable
変数捕獲	variable capture
変性	variance

ホ

ポインタ算術	pointer arithmetic
包摂	subsumption
飽和集合	saturated set
捕獲回避代入	capture-avoiding substitution
保存	preservation
ボックス化された	boxed

マ

マーシャリング	marshaling
前順序	preorder
交わり	meet
マッチング	matching
末尾再帰	tail-recursive
末尾呼び出し	tail call
マルチメソッド	multi-method

ミ

未知の基本型	unknown base type

ム

結び	join

メ

明示的代入	explicit substitution
命題＝型	propositions as types
メソッド呼び出し	method invocation
メタ言語	metalanguage
メタ数学	meta-mathematics
メタ変数	metavariables
メタ理論	metatheory

モ

モジュール	module
モジュールシステム	module systems
モジュール性	modularity
モジュラープログラミング	modular programming
モデル検査	model checking
モデル検査器	model checker
モナド	monad

ヤ

矢印型	arrow type

ユ

有界型演算子	bounded type operator
有界全称型	bounded universals
有界存在型	bounded existentials
有界量化	bounded quantification
有界量化子	bounded quantifier
有限オーバーロード	finitary overloading
優先順位	precedence

ヨ

様相論理	modal logic
余帰納法	coinduction
よく振る舞う	well behaved
より一般的（代入が）	more general
弱いポインタ	weak pointer

ラ

ラベル	label
ラムダキューブ	lambda cube
ラムダ計算	lambda-calculus
ラムダ項	lambda-term
ラムダ抽象	lambda-abstraction

リ

リージョン推論	region inference
リフレクション	reflection
略記/省略	abbreviation
領域理論	domain theory
リンク（モジュールの）	linking

ル

累積的	cumulative

レ

例外	exception
例外ハンドラ	exception handler
レキシカルスコープ	lexical scope
レコード	record
レコードのカインド	record kind
列	sequence
列カインド	row kind
列挙型	enumerated type/enumeration
列変数	row variable
列変数多相	row-variable polymorphism

ロ

露出している（項が/型が）	exposed
論理関係	logical relation
論理述語	logical predicate

ワ

ワイルドカード束縛子	wildcard binder
和型	sum type

規則図一覧

索引

記号・ギリシア文字・数字

A

B

C

D

E

F

キ

ク

ケ

ハ

ヒ

フ

ヘ

ホ

マ

ミ

ム

メ

モ

ヤ

ユ

ヨ

ラ

リ

レ

ロ

ワ

著者について

Benjamin C. Pierce

Pennsylvania 大学計算機・情報科学科教授（Salvatori Professor）。主な研究分野はプログラミング言語、静的型システム、ソフトウェアセキュリティ、並行・分散プログラミング、および同期技術。著書に "Types and Programming Languages"（本書）、"Software Foundations" などがあり、大学院の教科書として幅広く用いられている。多くの一般ユーザを持つファイル同期ソフトウェア Unison のメイン設計者でもある。

監訳者について

住井英二郎（すみいえいじろう）

1998 年東京大学理学部情報科学科卒業。2000 年 Pennsylvania 大学 Visiting Scholar。2001 年東京大学大学院情報理工学系研究科コンピュータ科学専攻助手（同情報学環流動教員）。2003 年 Pennsylvania 大学 Research Associate。2004 年東京大学大学院博士（情報理工学）。2005 年東北大学大学院情報科学研究科助教授（2007 年同准教授）、現在に至る。日本学術振興会賞、日本 IBM 科学賞、マイクロソフトリサーチ日本情報学研究賞等受賞。日本学術会議特任連携会員。未踏ソフトウェア創造事業スーパークリエータ。

訳者について

遠藤侑介（えんどうゆうすけ）

2006 年東京大学大学院情報理工学系研究科修士課程修了。現在、株式会社 東芝 研究開発センター勤務。プログラミング言語、分散データベースに関する研究開発に従事。趣味は Ruby プログラミング言語の開発、関東の鉄道の沿線ウォーキング。訳書に『抽象によるソフトウェア設計 — Alloy ではじめる形式手法』（共訳、オーム社）。

酒井政裕（さかいまさひろ）

2007 年慶應義塾大学大学院政策・メディア研究科修士課程修了。現在、株式会社 東芝 研究開発センター勤務。形式手法、ソフトウェアテストなどのソフトウェア高信頼化の研究に従事。関数的プログラミングや理論計算機科学にも興味を持つ。訳書に『抽象によるソフトウェア設計 — Alloy ではじめる形式手法』（共訳、オーム社）。

今井敬吾（いまいけいご）

2009 年名古屋大学大学院情報科学研究科博士後期課程単位取得退学。博士（情報科学）。2007 年 同研究科附属組込みシステム研究センター研究員、2010 年より有限会社 IT プランニング勤務。π 計算など並行計算の型システムに興味を持つ傍ら、実務では OCaml や Haskell を用いたソフトウェア開発を行う。

黒木裕介（くろきゆうすけ）

2007 年東京大学大学院情報理工学系研究科修士課程修了。現在、株式会社 東芝 研究開発センター勤務。スケジューリングなどの研究開発に従事（専門は数理最適化）。他に、TEX ユーザの集い 2009–2012 や TUG 2013（第 34 回 TEX Users Group 年次大会）で実行委員を務める。

今井宜洋（いまいよしひろ）

2007 年名古屋大学大学院多元数理科学研究科博士課程前期課程修了。現在、有限会社 IT プランニング勤務。関数型プログラミング言語や証明支援系 Coq を使ったシステム開発に従事。証明系に関する勉強会 ProofCafe を主催する他、トップエスイーの講師を務める。

才川隆文（さいかわたかふみ）

2011 年名古屋大学大学院多元数理科学研究科博士課程前期課程修了。現在、同後期課程在学。型理論など計算機科学と、関連する数学を研究。趣味は型無しの理論や言語。

今井健男（いまいたけお）

2000 年東京大学大学院理学系研究科修士課程修了。現在、株式会社 東芝 研究開発センター勤務。形式手法、ソフトウェアテストなどのソフトウェア高信頼化の研究に従事。訳書に『抽象によるソフトウェア設計 — Alloy ではじめる形式手法』（共訳、オーム社）。

- 本書の内容に関する質問は、オーム社開発部「＜書名を記載＞」係宛、E-mail (kaihatu@ohmsha.co.jp) または書状、FAX (03-3293-2825) にてお願いします。お受けできる質問は本書で紹介した内容に限らせていただきます。なお、電話での質問にはお答えできませんので、あらかじめご了承ください。
- 万一、落丁・乱丁の場合は、送料当社負担でお取替えいたします。当社販売課宛お送りください。
- 本書の一部の複写複製を希望される場合は、本書扉裏を参照してください。

型システム入門 プログラミング言語と型の理論

平成25年 3月25日　第1版第1刷 発行
平成26年 4月10日　第1版第2刷 発行

著　　者　Benjamin C. Pierce
監 訳 者　住井英二郎
訳　　者　遠藤侑介・酒井政裕・今井敬吾・黒木裕介・今井宜洋・才川隆文・今井健男
企画編集　オーム社 開発局
発 行 者　竹 生 修 己
発 行 所　株式会社 オ ー ム 社
郵便番号　101-8460
東京都千代田区神田錦町 3-1
電 話　03 (3233) 0641 (代表)
URL　http://www.ohmsha.co.jp/

印刷　エヌ・ピー・エス　　製本　三水舎
ISBN 978-4-274-06911-6　Printed in Japan